What is ActionScript and Why Should I Care?
This tutorial lesson is the first in a continuing series of lessons dedicated to teaching object-oriented programming (OOP) with ActionScript.

Table of Contents

Preface

This tutorial lesson is the first in a continuing series of lessons dedicated to teaching object-oriented programming *(OOP)* with ActionScript.

Note: Note that all references to ActionScript in this lesson are references to version 3 or later.

Developing ActionScript programs

There are several ways to develop programs using the ActionScript programming language. For most of the lessons in this series, I will use Adobe's Flash Builder 4 or the free FlashDevelop as an integrated development environment *(IDE)* .

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3, which was the predecessor to Flash Builder 4. *(See Baldwin's Flex programming website.)* You should read that lesson before embarking on the lessons in this series.

What is ActionScript?

According to the ActionScript Technology Center,

> "Adobe ActionScript is the programming language of the Adobe Flash Platform. Originally developed as a way for developers to program interactivity, ActionScript enables efficient programming of Adobe Flash Platform applications for everything from simple animations to complex, data-rich, interactive application interfaces.

> First introduced in Flash Player 9, ActionScript 3.0 is an object-oriented programming (OOP) language based on ECMAScript -- the same standard that is the basis for JavaScript -- and provides incredible gains in runtime performance and developer productivity."

What is the Adobe Flash Platform?

According to Adobe Flash Platform ,

> "The Adobe Flash Platform is an integrated set of technologies surrounded by an established ecosystem of support programs, business partners, and enthusiastic user communities. Together, they provide everything you need to create and deliver the most compelling applications, content, and video to the widest possible audience."

The primary delivery mechanisms for applications built with the Adobe Flash Platform are the Adobe Flash Player and Adobe Air .

What is the Adobe Flash Player?

According to Adobe Flash Player ,

> "Flash Player is a cross-platform browser plug-in that delivers breakthrough Web experiences to over 99% of Internet users."

What is Adobe Air?

According to Adobe Air ,

> "The Adobe AIR runtime lets developers use proven web technologies to build rich Internet applications that run outside the browser on multiple operating systems."

What about game programming?

As of November 2009, a significant portion of the game programming marketplace involved Flash games that are launched from Facebook and similar social networking sites. According to the Adobe Facebook page ,

> "The Adobe Flash Platform and Facebook Platform provide the ideal solution for building rich, social experiences on the web. Flash is available on more than 98% of Internet-connected PCs, so people can immediately access the applications, content, and video that enable social interactions. The Facebook Platform is

> used by millions of people every day to connect and share with the people in their lives. Together, both platforms allow you to:
>
> - **Share:** Create rich interactions for users to share with friends.
> - **Have fun:** Make games social; let users compete against their friends.
> - **Connect:** Let users connect to your RIAs with Facebook Connect.
> - **Solve problems:** Build RIAs that harness the power of community.
> - **Reach people:** Reach millions of Facebook users through social distribution.
>
> The new ActionScript 3.0 Client Library for Facebook Platform API, fully supported by Facebook and Adobe, makes it easy to build applications that combine the strengths of the Flash Platform and Facebook Platform."

What about iPhone programming

According to Adobe Labs Applications for iPhone ,

> "Flash Professional CS5 will enable you to build applications for iPhone and iPod touch using **ActionScript 3** . These applications can be delivered to iPhone and iPod touch users through the Apple App Store.*
>
> A public beta of Flash Professional CS5 with prerelease support for building applications for iPhone is planned for later this year."

Why should I care?

Adobe ActionScript is the programming language of the Adobe Flash Platform.

Applications developed with the Adobe Flash Platform are primarily delivered to the client via the Adobe Flash Player.

The Adobe Flash Player is a cross-platform browser plug-in that delivers Web experiences to over 98% of Internet users.

Flash Professional CS5 will enable you to build applications for iPhone and iPod touch using ActionScript 3.

Therefore, if you want to learn a programming language that allows you to tap into a market consisting of more than 98% of Internet users or if you want to learn a programming language that will enable you to build applications for iPhone and iPod touch, ActionScript may be the programming language for you.

Examples of Rich Internet Applications *(RIAs)*

Here are some websites that use Rich Internet Applications built with the Adobe Flash Platform *(November 2009)* :

- Adobe Flash Platform
- Adobe Flash Player
- Flash Platform and DIRECTV
- Flash Platform and Demandbase
- Flash Platform and Finetune
- Flash Platform and NASDAQ
- Flash Platform and Playfish
- Flash Platform and SAP BusinessObjects Xcelsius
- Flash Platform and Sharp Electronics
- CNN.com Live
- Fox News Live Stream
- NBC - WWW.CHANNELSURFING.NET

Live streaming TV feeds

There are many other live streaming TV feeds that were built with the Adobe Flash Platform. You can find them with a Google search.

Prerequisites for study

In the event that you will be studying this material for the purpose of learning about OOP with ActionScript, there are a few thing that you need to know.

Not for beginners

First, this is not a beginners programming course. In developing this material, I will assume that you already know how to program in some procedural programming language and that you already understand such fundamental concepts as programming logic, functions, parameter passing, etc.

Instead, this series of lessons will be designed to teach more advanced concepts involving object-oriented programming *(OOP)* using ActionScript.

ActionScript 3 or later will be required

The lessons in this series will emphasize OOP and will be based on ActionScript 3 or a later version. No attempt will be made to achieve backward compatibility with ActionScript 1 or ActionScript 2.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: What is ActionScript and Why Should I care?
- Files:
  - ActionScript0102\ActionScript0102.htm
  - ActionScript0102\Connexions\ActionScriptXhtml0102.htm

-end-

What is OOP and Why Should I Care?
A very brief introduction to the concepts of object-oriented programming with descriptions of encapsulation, inheritance, and polymorphism. Includes two new custom components that will run in the Flash player plugin.

Table of Contents

Preface

General

Note: Note that all references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a continuing series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for ActionScript programs. An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should read that lesson before embarking on the lessons in this series.

One of the complicating factors

One of the complicating factors in using Adobe Flex as the launch pad for ActionScript programs is the difficulty of understanding the relationships that exist between Flex and ActionScript. I recommend that you study the following lessons on Baldwin's Flex programming website that address that topic:

- 114 Integrating ActionScript and Flex
- 116 Defining Custom MXML Components
- 118 Defining Custom ActionScript Components
- 120 Creating Online Tests using Custom ActionScript Components

I recommend that you also study all of the other lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both technologies and the relationships that exist between them in order to become a successful ActionScript programmer.

Another complicating factor

Another complicating factor is knowing whether to use ActionScript code or Flex MXML code to achieve a particular objective. In many cases, either will do the job.

Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

What is object-oriented programming (OOP)?

If you Google this question, you will get hundreds of answers. Here is my answer along with an anecdotal description.

Unlike earlier programming styles,

> *Object-oriented programming is a programming style that mimics the way most people think and work .*

An anecdotal description

If you have ever assembled a child's playscape in your back yard, this scenario should sound familiar to you.

When you opened the large boxes containing the playscape, hundreds of objects spilled onto the ground. Those objects may have consisted of braces, chains, swing seats, slides, screws, nuts, bolts, washers, climbing ropes, ladder rungs, and other assorted objects.

Atomic and non-atomic objects

I will refer to the kind of object that I have described in the above list as atomic objects. What I mean by that is that they can't be easily subdivided into smaller objects.

If you were lucky, some of the objects in the box may not have been atomic objects. Instead they may have been pre-assembled arrangements of atomic objects such as a climbing net composed of individual ropes tied together to form a net.

Your job - assemble the objects

Your job was to assemble those hundreds of atomic and non-atomic objects into a final object which you proudly referred to as *"The Playscape."*

Objects working together

It has been said that a successful object-oriented program consists of a bunch of cooperating software objects working together to achieve a specified behavior. The overall behavior of the program is the combination of behaviors of the individual objects. For example, some objects may acquire input data, other objects may compute and produce output data, while other objects may display the output data.

It could also be said that a playscape consists of a bunch of hardware objects working together to achieve a specified behavior. The overall behavior of the playscape is the combination of behaviors of the individual objects. For example, the behavior of some of the braces is to stand strong and not bend or break, while the behavior of a swing is to be flexible and move in a prescribed way.

Creating a model

One of the tasks of an object-oriented programmer is to assemble software objects into a model that often represents something that exists in the real world. For a very visual example, you might be asked to create an advertising web page showing an animated software model of the playscape that you assembled in your back yard. With the playscape, you were simply required to assemble the existing hardware objects. However, in the object-oriented programming world, you must do more than just assemble objects.

Objects must be designed and manufactured

Getting back to the playscape, every one of the objects for the playscape was manufactured before being shipped to you. Even before that, each object was designed by someone and a set of manufacturing drawings was probably created so that the object could be mass produced in a high-volume manufacturing facility.

A class is analogous to manufacturing drawings

In OOP, there is a direct analogy to the manufacturing drawings of the hardware world. We call it a **class** . A class documents the specifications for the construction of a particular type of software object.

A large library of classes

As an object-oriented programmer, you will typically have access to a large library of existing classes from which you can construct different types of software objects, such as buttons, sliders, etc. In addition, you will often need to design and define new classes from which you can construct new types of objects.

Why should I care?

Although ActionScript and Flex provide a large class library from which you can construct objects *(components)* , you will probably need to create new custom components from time to time as well.

ActionScript is usually required

Flex/MXML can be used to create simple custom components but ActionScript is often needed to cause those components to have more interesting behavior.

Sometimes you can embed or include non-OO ActionScript code in a Flex MXML file to achieve the desired behavior. Often, however, you will need to create your new component almost entirely in ActionScript. You will need to understand OOP in order to do that.

Start from scratch or extend an existing component class

An understanding of OOP is particularly important if you need to create a new component from scratch or create a new component by modifying the appearance and/or behavior of a component for which a class already exists.

I will provide an example of a new custom component that extends an existing component later in this lesson.

Therefore, if you plan to create rich internet applications, games, or iPhone applications using ActionScript, you will often need to understand OOP in order to create custom components that your clients find interesting.

General background information

Some languages such as C do not readily support OOP. Other languages such as C++ and ActionScript support OOP, but don't require you to use the object-oriented features of the language. Still other languages such as Java and C# require you to program using OOP techniques.

OOP is not enforced

Because ActionScript does not enforce a requirement for your code to be object oriented *(OO)* , it is possible to learn to use major aspects of ActionScript without ever learning to use the object-oriented features. This approach simply requires you to learn how to use the rudimentary aspects of the language.

The real challenge

The real challenge to becoming an ActionScript programmer is not simply to learn the rudimentary aspects of the language. The real challenge lies in:

- Learning to productively use the large class library provided as part of the software development kit.
- Learning to design and define new classes when needed.
- Learning to design and program in the object-oriented paradigm.

Learn the library gradually

The first of these challenges can be met on a gradual basis. In other words, it is not necessary to memorize the entire class library to produce useful OO programs. However, it is necessary to learn how to use the library documentation to find what you need.

Some things can't be learned gradually

The remaining two challenges cannot easily be met on a gradual basis. Many aspects of OOP must be understood before a programmer can successfully write OO programs

A slightly more technical description of OOP

An introductory description of OOP can be based on the guideline in Figure 1.
Guideline for OOP.

Figure

The solution to the problem should resemble the problem, and observers of the solution should be able to recognize the problem without necessarily knowing about it in advance.

Guideline for OOP.

For example, an OO program that deals with banking transactions should be recognizable on the basis of the objects that it uses, such as deposit objects, withdrawal objects, account objects, etc.

Flex is a good example

Many "application frameworks" are written according to the OOP paradigm. Important among these is Adobe's Flex, which can be used to simplify the development of the graphical user interface *(GUI)* portions of ActionScript programs.

Flex makes it possible to use uncomplicated XML syntax to access and use the ActionScript class library. Adobe's Flash Builder 4 makes the process even more straightforward by providing a largely drag-and-drop visual development environment for creating the GUI portion of ActionScript programs.

All of the components that are available in Flex are objects created from classes in the ActionScript library. Those classes have names like **Button** , **RadioButton** , and **NumericStepper** *(see Figure 2)* .

Three important concepts

Any object-oriented language must support three very important concepts:

- Encapsulation,
- Inheritance,
- Polymorphism.

We use these three concepts extensively as we attempt to model the real-world problems that we are trying to solve with our object-oriented programs. I will provide brief descriptions of these concepts in the remainder of this lesson and explain them in detail in future lessons.

Encapsulation example

Consider the steering mechanism of a car as a real-world example of **encapsulation.** During the past eighty years or so, the steering mechanism for the automobile has evolved into an *object* in the OOP sense.

Only the interface is exposed

In particular, most of us know how to use the steering mechanism of an automobile without having any idea whatsoever how it is implemented. All most of us care about is the *interface* , which we often refer to as a steering wheel. We know that if we turn the steering wheel clockwise, the car will turn to the right, and if we turn it counterclockwise, the car will turn to the left.

How is it implemented?

Most of us don't know, and don't really care, how the steering mechanism is actually implemented *"under the hood."* In fact, there are probably a number of different implementations for various brands and models of automobiles. Regardless of the brand and model, however, the human interface is pretty much the same. Clockwise turns to the right, counterclockwise turns to the left.

As in the steering mechanism for a car, a common approach in OOP is to *"hide the implementation"* and *"expose the interface"* through *encapsulation.*

Inheritance example

Another important aspect of OOP is *inheritance* . Let's form an analogy with the teenager who is building a hotrod. That teenager doesn't normally start with a large chunk of steel and carve an engine out of it. Rather, the teenager will usually start with an existing engine and make improvements to it.

In OOP lingo, that teenager *extends* the existing engine, *derives* from the existing engine, *inherits* from the existing engine, or *subclasses* the existing engine *(depending on which author is describing the process)* .

Just like in *"souping up"* an engine for a hotrod, a very common practice in OOP is to create new improved objects by extending existing class definitions.

Reuse, don't reinvent

One of the major arguments in favor of OOP is that it provides a formal mechanism that encourages the reuse of existing programming elements. One of the mottos of OOP is *"reuse, don't reinvent."*

Polymorphism example

A third important aspect of OOP is *polymorphism* . This is a Greek word meaning something like *one name, many forms* . This is a little more difficult to explain in non-programming terminology. However, we will stretch our imagination a little and say that polymorphism is somewhat akin to the automatic transmission in your car. In my Honda, for example, the automatic transmission has four different *methods* or *functions* known collectively as *Drive (in addition to the functions of Reverse, Park, and Neutral)* .

Select Drive to go forward

As an operator of the automobile, I simply select *Drive (meaning go forward)* . Depending on various conditions at *runtime* , the automatic transmission system decides which version of the *Drive* function to use in every specific situation. The specific version of the function that is used is based on the current conditions *(speed, incline, etc.)* . This is somewhat analogous to what we will refer to in a subsequent tutorial lesson as *runtime polymorphism* .

Object-oriented programming vocabulary

OOP involves a whole new vocabulary *(or jargon)* which is different from or supplemental to the vocabulary of procedural programming.

For example the object-oriented programmer defines an *abstract data type* by *encapsulating* its *implementation* and its *interface* into a *class* .

One or more *instances* of the class can then be *instantiated* .

An *instance* of a class is known as an *object* .

Every *object* has *state* and *behavior* where the *state* is determined by the current values stored in the object's *instance variables* and the *behavior* is determined by the *instance methods* of the class from which the *object* was *instantiated* .

Inherited abstract data types are *derived classes* or *subclasses* of *base classes* or *super classes* . We *extend super classes* to create *subclasses* .

Within the program, the code *instantiates objects (creates instances of classes)* and sends *messages* to the *objects* by invoking or calling the class's *methods (or member functions)* .

If a program is "object oriented", it uses *encapsulation* , *inheritance* , and *polymorphism* . It defines *abstract data types* , *encapsulates* those abstract data types into *classes* , *instantiates objects* , and *sends messages* to the *objects* .

To make things even more confusing, almost every item or action used in the OOP jargon has evolved to be described by several different terms. For example, we can cause an object to *change its state* by *sending it a message* , *calling its methods* , or *calling its member functions* . The term being used often depends on the author who wrote the specific book that you happen to be reading at the time.

Hopefully most of this terminology will become clear as we pursue these lessons.

Sample code

We have reached the point in this discussion where I should provide some examples of ActionScript 3 class definitions so that you can get a preview of what lies ahead.

A simple class named MyClass

The code beginning with the word **public** in Listing 1 is the definition of a very simple class named **MyClass** . I don't expect this code to mean much to you at this point. Suffice it to say that an object of this class is a new GUI component consisting of a **Button** object, a **RadioButton** object, and a **NumericStepper** object, all contained in a **VBox** object.

Example:

Class file named MyClass.as.

```
package Classes{
  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.RadioButton;
  import mx.controls.NumericStepper;

  public class MyClass extends VBox{
    //Constructor follows
    public function MyClass(){
      addChild(new Button());
      addChild(new RadioButton());
      addChild(new NumericStepper());
      setStyle("backgroundColor",0xFFFF00);
    }//end constructor
  }//end class
}//end package
```

A new data type

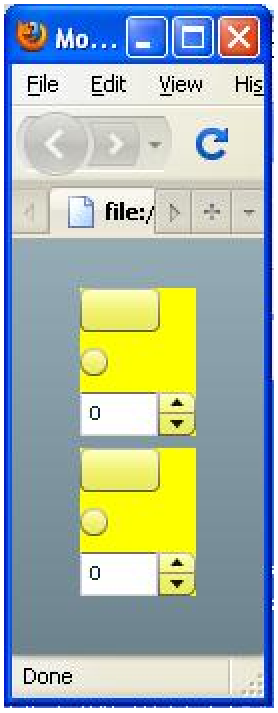
The definition of the class creates a new data type, which is unknown to the compiler until it is defined by the programmer.

Two instances *(objects)* of the class named MyClass

Two objects of the new class named **MyClass** are shown in Figure 2. You should be able to spot the **Button** object *(rectangular)* , the **RadioButton** object *(circular)* , and the **NumericStepper** object in each of the two **MyClass** objects.

Two objects of the class named MyClass.

Figure



Two objects of the class named MyClass.

A yellow background

Normally you can't see a **VBox** object. You can only see the components that it contains. However, I included code in Listing 1 to set the background style of the **VBox** object to yellow to make it possible for you to visually separate the two new custom objects in Figure 2.

The Flex MXML file

Listing 2 shows the Flex file that was used to instantiate the two objects of the class named **MyClass** and to display them in the Flash Player window in a browser.

Example:
Flex file named SimpleClass01.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<!--SimpleClass01 -->

<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:classes="Classes.*">

    <classes:MyClass />
    <classes:MyClass />

</mx:Application>
```

The two lines of code beginning with the words **classes** in Listing 2 cause the two new objects to be instantiated and displayed.

Another custom component - NumericTextAreaA

One of the standard Flex components is named **TextArea** . This component is an object of the **ActionScript** class having the same name.

A custom component that extends TextArea

As an example of creating a new custom component that extends an existing class, I will present and discuss a new custom component that extends the **TextArea** class. The new class is named **NumericTextAreaA.**

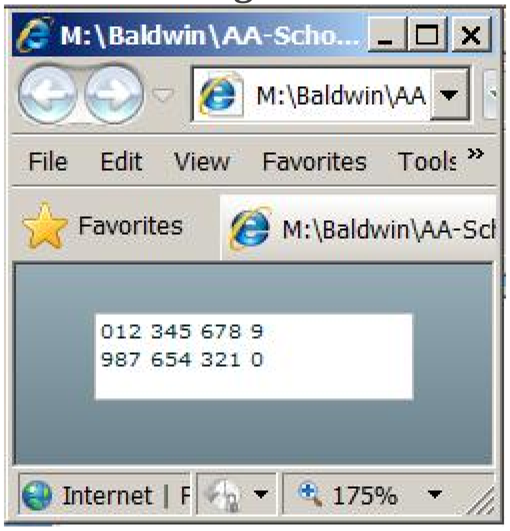
An object of the **NumericTextAreaA** class behaves just like an object of the standard **TextArea** class except that the new component will only accept numeric characters, the space character, the backspace character, and the return character. All other characters are rejected when the user attempts to type them into the text area.

An object of the class named NumericTextAreaA

Figure 3 shows the visual manifestation of an object of this class with two lines of numeric and space characters having been entered. *(The entry of additional lines of text causes scroll bars to automatically appear.)*

An object of the class named NumericTextAreaA.

Figure



```
012 345 678 9
987 654 321 0
```

An object of the class named NumericTextAreaA.

A class definition is probably required

It is possible to create simple custom components by embedding ActionScript code in a Flex MXML file. However, I don't believe that it is possible to create an object with this behavior without defining a new ActionScript class. Even if it is possible, defining a new class is the approach that makes the most sense to me.

The file named NumericTextArea01.mxml

The MXML code beginning with **cc** in Listing 3 instantiates the object shown in Figure 3.

Example:
The file named NumericTextArea01.mxml.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
    xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:cc="CustomClasses.*">

    <cc:NumericTextAreaA/>

</mx:Application>
```

The class definition for NumericTextAreaA

Listing 4 shows the class definition from which the object shown in Figure 3 was instantiated.

Example:
The class definition for NumericTextAreaA.

```
/*This is a custom component. This class extends
the
TextArea class.  It recognizes only the following
characters:
0, 1, 2, 3, 4, 5, 6, 7, 8,9
space
backspace
return key
```

```
The next numeric character is appended onto the
end of
the string in the text area regardless of the
current
position of the cursor.

The backspace key deletes characters from the end
of the
string, one character at a time.
**************************************************
*******/

package CustomClasses{
    import mx.controls.TextArea;
    import mx.controls.Alert;
    import flash.events.*

    public class NumericTextAreaA extends
TextArea{
        private var theText:String = "";

        public function NumericTextAreaA(){

this.addEventListener("keyUp",processKey);
        }//end constructor

        private function processKey(

event:KeyboardEvent):void{
            if(event.charCode==48){
                theText+="0";
            }else if(event.charCode==49){
                theText+="1";
            }else if(event.charCode==50){
                theText+="2";
            }else if(event.charCode==51){
                theText+="3";
            }else if(event.charCode==52){
                theText+="4";
            }else if(event.charCode==53){
                theText+="5";
            }else if(event.charCode==54){
                theText+="6";
            }else if(event.charCode==55){
                theText+="7";
            }else if(event.charCode==56){
                theText+="8";
            }else if(event.charCode==57){
                theText+="9";
            }else if(event.charCode==8)
{//backspace
                theText=

theText.substr(0,theText.length-1);
            }else if(event.charCode==32){//space
                theText+=" ";
            }else if(event.charCode==13){//return
key
                theText+="\n";
            }//end else

            this.text=theText;
        }//end processKey method
    }//end class
}//end package
```

May not be familiar code

There may be quite a lot of code in Listing 4 with which you are not familiar. However, it is not my purpose in writing this lesson to get into the details of defining classes in ActionScript, so I won't take the time to explain this code in this lesson. Suffice it to say that Listing 4 checks the character code associated with each keystroke in the text area and rejects all but the numeric characters, the backspace character, the space character, and the return key.

I will explain code like this in detail in future lessons. For now, simply accept this as an example of why you may need to learn OOP in order to advance your career as an ActionScript programmer.

Running the ActionScript program named NumericTextArea01

If you have the Flash Player plug-in (version 9 or later) installed in your browser you should be able to run this program by clicking on NumericTextArea01 .

> *Enter some alphabetic and numeric text in the white box to see how the GUI component behaves. Click the "Back" button in your browser to return to this page when you are finished experimenting with the component.*

If you don't have the proper Flash Player installed, you should be notified of that fact and given an opportunity to download and install the Flash Player plug-in program.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Miscellaneous

Note: Housekeeping material

- Module name: What is OOP and Why Should I Care?
- Files:
    - ActionScript0104\ActionScript0104.htm
    - ActionScript0104\Connexions\ActionScriptXhtml0104.htm

-end-

Encapsulation - The Big Picture
Learn two different approaches for using ActionScript to create complex custom components having the same programming interface, the same user interface, and the same behavior.

Note: Click Encapsulation01 to run this ActionScript program. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

Note: Note that all references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript .

The three main characteristics of an object-oriented program

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

In this and the next two lessons, I will explain and illustrate these three characteristics from a big-picture viewpoint. Following that, I will get down in the weeds and start explaining in detail how to use ActionScript for object-oriented programming *(OOP)* .

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

In addition to the three explicit characteristics of *encapsulation* , *inheritance* , and *polymorphism* , an object-oriented program also has an implicit characteristic of *abstraction* .

What is abstraction?

Abstraction is the process by which we specify a new data type, often referred to an abstract data type or ADT.

How does abstraction relate to encapsulation?

Encapsulation is the process of gathering an ADT's *data representation* and *behavior* into one encapsulated entity. In other words, encapsulation converts from the abstract to the concrete.

Some analogies

You might think of this as being similar to converting an idea for an invention into a set of blueprints from which it can be built, or converting a set of written specifications for a widget into a set of drawings that can be used by the machine shop to build the widget.

Automotive engineers encapsulated the specifications for the steering mechanism of my car into a set of manufacturing drawings. Then manufacturing personnel used those drawings to produce an object where they exposed the interface *(steering wheel)* and hid the implementation *(levers, bolts, etc.)* .

In all likelihood, the steering mechanism object contains a number of other more-specialized embedded objects, each of which has *state* and *behavior* and also has *an interface* and *an implementation* .

The interfaces for those embedded objects aren't exposed to me, but they are exposed to the other parts of the steering mechanism that use them.

Abstraction

Abstraction is the specification of an abstract data type, which includes a specification of the type's *data representation* and its *behavior* . In particular,

- What kind of data can be stored in an entity of the new type, and
- What are all the ways that the data can be manipulated?

A new type

For our purposes, an abstract data type is a new type *(not intrinsic to the ActionScript language)* . It is not one of the primitive data types that are built into the programming language *(such as Boolean, int, Number, String, and uint)* .

Already known to the compiler

The distinction in the previous paragraph is very important. The data representation and behavior of the intrinsic or primitive types is already known to the compiler and cannot normally be modified by the programmer.

Not known to the compiler

The representation and behavior of an abstract type is not known to the compiler until it is defined by the programmer and presented to the compiler in an appropriate manner.

Define data representation and behavior in a class

ActionScript programmers define the *data representation* and the *behavior* of a new type *(present the specification to the compiler)* using the keyword **class** . In other words, the keyword **class** is used to convert the specification of a new type into something that the compiler can work with; a *set of plans* as it were. To define a class is to go from the abstract to the concrete.

Create instances of the new type

Once the new type *(class)* is defined, one or more objects of that type can be brought into being *(instantiated, caused to occupy memory)* .

Objects have state and behavior

Once instantiated, the object is said to have *state* and *behavior* . The *state* of an object is determined by the current values of the data that it contains and the *behavior* of an object is determined by its methods.

The state and behavior of a GUI Button object

For example, if we think of a GUI **Button** as an object, it is fairly easy to visualize the object's state and behavior.

A GUI Button can usually manifest any of a number of different states: size, position, depressed image, not depressed image, label, etc. Each of these states is determined by data stored in the *instance variables* of the **Button** object at any given point in time. *(The combination of one or more instance variables that determine a particular state is often referred to as a property of the object.)*

Similarly, it is not too difficult to visualize the behavior of a GUI **Button** . When you click it with the mouse, some specific action usually occurs.

An ActionScript class named Button

If you dig deeply enough into the ActionScript class library, you will find that there is a class named **Button** . Each individual **Button** object in a Flex application is an instance of the ActionScript class named **Button** .

The state of Button objects

Each **Button** object has instance variables, which it does not share with other **Button** objects. The values of the instance variables define the *state* of the button at any given time. Other **Button** objects in the same scope can have different values in their instance variables. Hence they can have a different state.

The behavior of a Button object

Each Button object also has certain fundamental behaviors such as responding to a mouse **click** event or responding to a **mouseOver** event.

The ActionScript programmer has control over the code that is executed in response to the event. However, the ActionScript programmer has no control over the fact that a **Button** object will respond to such an event. The fact that a **Button** will respond to certain event types is an inherent part of the type specification for the **Button** class and can only be modified by modifying the source code for the **Button** class.

Encapsulation

If abstraction is the design or specification of a new type, then encapsulation is its definition and implementation.

A programmer defines the data representation and the behavior of an abstract data type into a class, thereby defining its implementation and its interface. That data representation and behavior is then encapsulated in objects that are instantiated from the class.

Expose the interface and hide the implementation

According to good object-oriented programming practice, an encapsulated design usually exposes the interface and hides the implementation. This is accomplished in different ways with different languages.

Just as most of us don't usually need to care about how the steering mechanism of a car is implemented, a user of a class should not need to care about the details of implementation for that class.

The user of the class *(the using programmer)* should only need to care that it works as advertised. Of course this assumes that the user of the class has access to good documentation describing the interface and the behavior of objects instantiated from the class.

Should be able to change the implementation later

For a properly designed class, the class designer should be able to come back later and change the implementation, perhaps changing the type of data structure used to store data in the object, and the using programs should not be affected by the change.

Class member access control

Object-oriented programming languages usually provide the ability to control access to the members of a class. For example, ActionScript, C++ and Java all use the keywords **public** , **private** , and **protected** to control access to the individual members of a class. In addition, ActionScript and Java add a fourth level of access control, which is called **internal** in ActionScript and is called **package-private** in Java. *(See Class property attributes in a companion document on ActionScript Resources.)*

Public, private, and protected

To a first approximation, you can probably guess what **public** and **private** mean. Public members are accessible by all code that has access to an object of the class. Private members are accessible only by members belonging to the class.

The **protected** keyword is used to provide inherited classes with special access to the members of their base classes.

A public user interface

In general, the user interface for a class consists of the **public** methods. *(The variables in a class can also be declared public but this is generally considered to be bad programming practice unless they are actually constants.)*

For a properly designed class, the class user stores, reads, and modifies values in the object's data by calling the **public** methods on a specific instance *(object)* of the class. *(This is sometimes referred to as sending a message to the object asking it to change its state)* .

ActionScript has a special form of method, often called an *implicit setter* method or an *implicit getter* method that is specifically used for this purpose. *(You will see several implicit setter methods in the program that I will explain later in this lesson.)*

Normally, if the class is properly designed and the implementation is hidden, the user cannot modify the values contained in the instance variables of the object without going through the prescribed public methods in the interface.

Not a good design by default

An object-oriented design is not a good design by default. In an attempt to produce good designs, experienced object-oriented programmers generally agree on certain design standards for classes. For example, the data members *(instance variables)* are usually **private** unless they are constants. The user interface usually consists only of **public** methods and includes few if any data members.

Of course, there are exceptions to every rule. One exception to this general rule is that data members that are intended to be used as symbolic constants are made public and defined in such a way that their values cannot be modified.

The methods in the interface should control access to, or provide a pathway to the private instance variables.

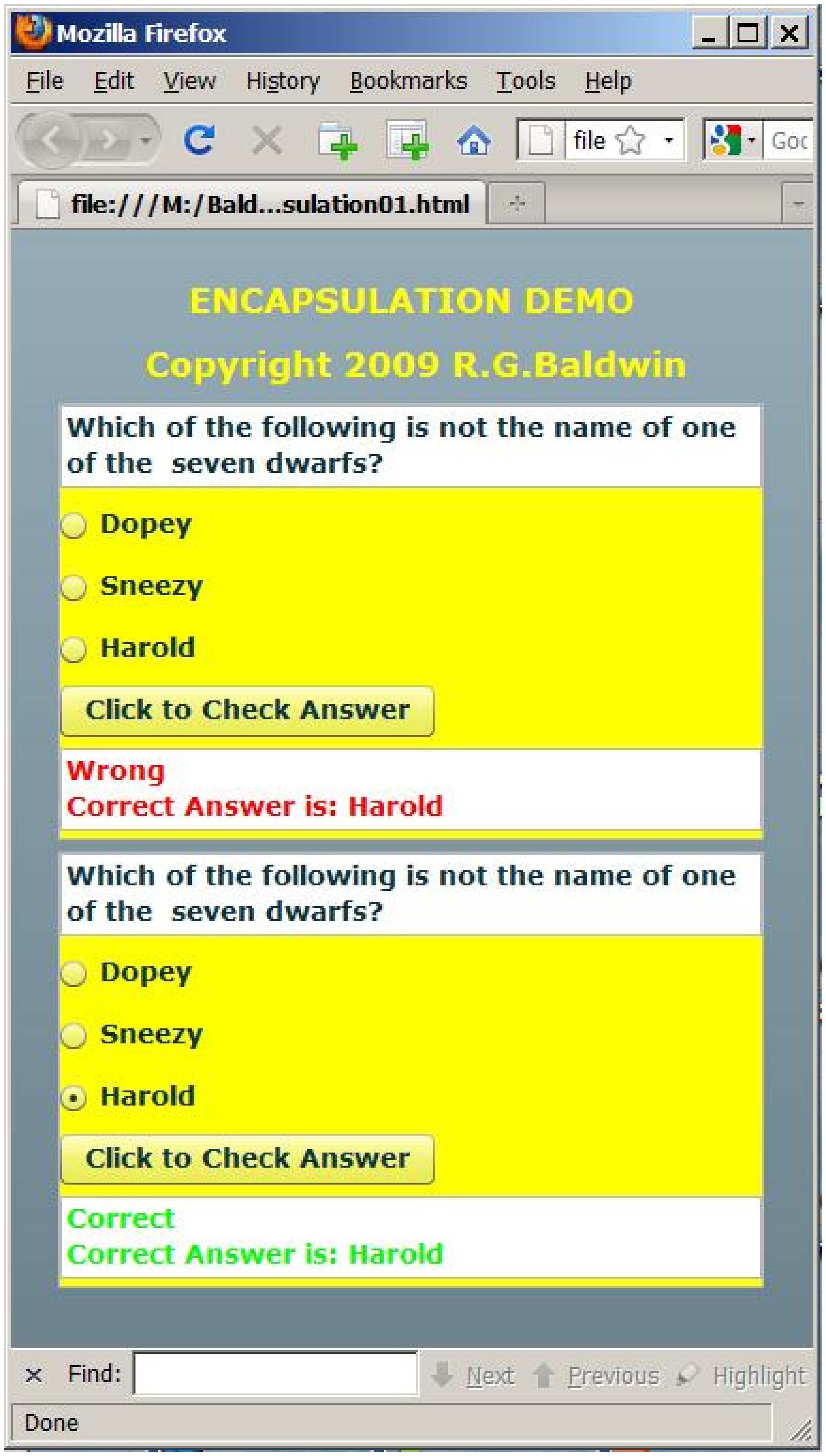
Not bound to the implementation

The interface should be generic in that it is not bound to any particular implementation. Hence, the class author should be able to change the implementation without affecting the using programs so long as the interface doesn't change.

In practice, this means that the signatures of the interface methods should not change, and that the interface methods and their arguments should continue to have the same meaning.

Preview

In this lesson, I will present and briefly explain a program with an output that consists of a Flex user interface containing two custom component objects as shown in Figure 1. Each of the objects is intended to represent a single multiple-choice question in an online test. *(See **Creating Online Tests using Custom ActionScript Components** here for a detailed explanation of the code.)*
Screen output for Encapsulation01.
Figure


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///M:/Bald...sulation01.html
Find: Next Previous Highlight
Done

ENCAPSULATION DEMO

Copyright 2009 R.G.Baldwin

Which of the following is not the name of one of the seven dwarfs?

( ) Dopey

( ) Sneezy

( ) Harold

Click to Check Answer

Wrong
Correct Answer is: Harold

Which of the following is not the name of one of the seven dwarfs?

( ) Dopey

( ) Sneezy

[✓ Harold]

Click to Check Answer

Correct
Correct Answer is: Harold

Screen output for Encapsulation01.

The two question objects in Figure 1 have the same behavior. And as you will see later, the classes from which they were instantiated have the same user interfaces. However, the classes are implemented in significantly different ways.

Discussion and sample code

Will discuss in fragments

I will discuss the code in this lesson in fragments. Listing 1 shows the mxml code that instantiates the two component objects shown in Figure 1. A complete listing of the file named **Encapsulation01.mxml** is provided in Listing 12 near the end of the lesson.

Example:
Mxml code to instantiate two custom component objects.

```
<!--The following code instantiates an object of
the class
named QuizA for a multiple-choice quiz question
with three
choices.-->

<cc:QuizA
question=
"Which of the following is not the name of one of
the
seven dwarfs?"
choice0="Dopey"
choice1="Sneezy"
choice2="Harold"
answer="2"
/>
```

```
<!--The following code instantiates an object of
the class
named QuizB for a multiple-choice quiz question
with three
choices. Note that the interface is exactly the
same as
for the class named QuizA. However, the
implementation of
QuizB is radically different from QuizA.-->

<cc:QuizB
question=
"Which of the following is not the name of one of
the
seven dwarfs?"
choice0="Dopey"
choice1="Sneezy"
choice2="Harold"
answer="2"
/>
```

The important thing...

The important thing to note in Listing 1 is that, with the exception of the name of the class being instantiated in each case *(QuizA and QuizB)* , the mxml code is identical for the two cases.

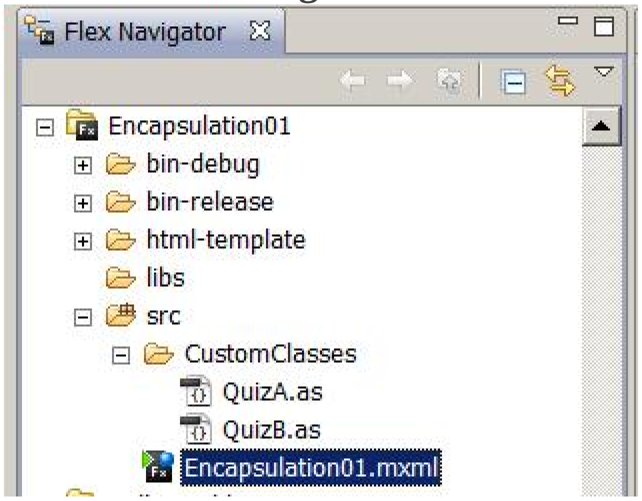
The code that begins with **cc:Quiza** produces the top question object in Figure 1 and the code that begins with **cc:Quizb** produces the bottom question object in Figure 1.

Since the mxml code for the two objects is identical, the user interface for the two classes must also be identical.

The project file structure

The Flex project file structure that I used for this program is fairly typical. However, before getting into a discussion of the two class files, I will show you how the files are organized in the Flex project as shown in Figure 2. File structure for the project named Encapsulation01.

Figure



File structure for the project named Encapsulation01.

Classes named QuizA and QuizB

As you can see from Listing 1 and Figure 2, two class files named **QuizA.as** and **QuizB.as** were used to instantiate the two objects shown in Figure 1. Complete listings for those two files are provided in Listing 13 and Listing 14 near the end of the lesson.

Remember, this is a big-picture discussion

Because this is a *"big-picture"* lesson, I won't explain either of these classes in detail in this lesson. *(You can find technical details for a class very similar to* ***QuizA*** *in* ***Creating Online Tests using Custom ActionScript Components*** *here you are interested in technical details at this point.)* Instead, I will compare the two class definitions from a big-picture viewpoint.

For brevity, I will also delete some of the code such as import directives.

Beginning of the class named QuizA

The class named **QuizA** begins in Listing 2.

Example:
Beginning of the class named QuizA.

```
package CustomClasses{
    //Import directives deleted for brevity.

  public class QuizA extends VBox{
    private var theQuestion:TextArea;
    private var choice00:RadioButton;
    private var choice01:RadioButton;
    private var choice02:RadioButton;
    private var checkButton:Button;
    private var result:TextArea;

    private var theAnswer:String;//numeric string
    private var correctAnswer:String;//actual
string
    private var vboxWidth:int = 375;
```

Variable declarations

The important thing to note in Listing 2 is the declaration of six instance variables of three different component types *(beginning with the first line that reads* ***private var*** *)* . These variables will be used to hold references to the six different components shown in each question object in Figure 1.

TextArea, RadioButton, and Button objects

The white rectangular areas at the top and the bottom of each question object in Figure 1 is an object of the class named **TextArea** . You can probably spot the three **RadioButton** objects and the **Button** object in each question object.

Beginning of the class named QuizB

The class named **QuizB** begins in Listing 3.

Example:
Beginning of the class named QuizB.

```
package CustomClasses{
    //Import directives deleted for brevity.

  public class QuizB extends VBox{
    private var components:Array =
        new Array(new TextArea(),//theQuestion
                  new RadioButton(),
                  new RadioButton(),
                  new RadioButton(),
                  new Button(),//checkButton
                  new TextArea());//result

    private var theAnswer:String;//numeric string
    private var correctAnswer:String;//actual
string
    private var vboxWidth:int = 375;
```

An array with six elements

The six instance variables that I referred to in Listing 2 were replaced by a single array having six elements in Listing 3. The creation of the array begins with the first line in Listing 3 that reads **private var** .

This is the major change that was made in the implementation of **QuizB** relative to the implementation of **QuizA** . This change will have significant ramifications throughout the remainder of the code whenever it is necessary to access a reference that points to one of the six components.

Populated with six component objects

It is also worth noting that the array elements are populated with references to component objects for **QuizB** when the array is created in Listing 3. The new component objects aren't instantiated until later in **QuizA** .

Implicit setter methods for QuizA

Listing 4 shows five implicit setter methods for the class named **QuizA** .

Example:
Implicit setter methods for QuizA.

```
    public function set
question(textIn:String):void{
        theQuestion.text = textIn;
    }//end implicit setter

    public function set
answer(answerIn:String):void{
        theAnswer = answerIn;
    }//end implicit setter

    public function set
choice0(choice:String):void{
        choice00.label=choice;
    }//end implicit setter

    public function set
```

```
choice1(choice:String):void{
        choice01.label=choice;
    }//end implicit setter

    public function set
choice2(choice:String):void{
        choice02.label=choice;
    }//end implicit setter
```

Setter methods are called by mxml code

Briefly, these methods are called by the code in Listing 1 when values are assigned to the following five mxml attributes:

- question
- choice0
- choice1
- choice2
- answer

See **Defining Custom MXML Components** here if you are interested in learning more about implicit setter methods at this point in time.

Implicit setter methods for QuizB

Listing 5 shows five implicit setter methods for **QuizB** that serve the same purpose as the five implicit setter methods for **QuizA** . Note the differences in the code that results from using individual variables to reference the components in **QuizA** and using an array to reference the components in **QuizB** .

Example:
Implicit setter methods for QuizB.

```
    public function set
question(textIn:String):void{
        components[0].text = textIn;
    }//end implicit setter

    public function set
answer(answerIn:String):void{
        theAnswer = answerIn;
    }//end implicit setter

    public function set
choice0(choice:String):void{
        components[1].label=choice;
    }//end implicit setter

    public function set
choice1(choice:String):void{
        components[2].label=choice;
    }//end implicit setter

    public function set
choice2(choice:String):void{
        components[3].label=choice;
    }//end implicit setter
```

Expose the interface but hide the implementation

These five setter methods, along with the class constructors, constitute the entire user interface for each class. If you examine Listing 13 and Listing 14, you will see that these five setter methods and the constructor are the only **public** members of either class. All other members of the classes are declared **private** .

Furthermore:

- The names of the five methods are the same in both classes.
- The names and types of the required parameters for the five methods are the same in both classes.
- The five methods serve the same purpose in both classes.
- The ultimate behavior of objects instantiated from the two classes is the same.

Therefore, the exposed user interface is the same for both classes but the hidden implementation is significantly different between the two classes.

Purpose of the setter methods

The purpose of the setter methods in both cases is to store mxml attribute values in the **text** property of the **TextArea** at the top of each question object in Figure 1 and to store mxml attribute values in the **label** property of each of the **RadioButton** objects in each question object in Figure 1. In addition, one of the setter methods stores an attribute value in the variable named **theAnswer** .

If you compare the code in Figure 4 and Figure 5, you will see that Figure 4 stores the incoming parameter values by way of the contents of four instance variables whereas Figure 5 stores the incoming parameter values by way of the contents of four elements in the array.

Constructor for QuizA

Listing 6 shows the constructor for the class named **QuizA** .

Example:
Constructor for QuizA.

```
    public function QuizA(){//constructor
      width=vboxWidth;
      setStyle("borderStyle","solid");
      setStyle("backgroundColor",0xffff00);
      theQuestion = new TextArea();
      theQuestion.editable = false;
      theQuestion.width=vboxWidth - 2;
      addChild(theQuestion);

      choice00 = new RadioButton();
      choice00.groupName="radioButtonGroup";
      addChild(choice00);

      choice01 = new RadioButton();
      choice01.groupName="radioButtonGroup";
      addChild(choice01);

      choice02 = new RadioButton();
      choice02.groupName="radioButtonGroup";
      addChild(choice02);

      checkButton = new Button();
      checkButton.label = "Click to Check Answer";

checkButton.addEventListener(MouseEvent.CLICK,

checkButtonHandler);
      addChild(checkButton);

      result = new TextArea();
      result.editable = false;
      result.width=vboxWidth - 2;
      result.visible=false;
      addChild(result);

      //Register an event listener that will be
      // executed when this object has been fully
      // constructed. It will set the height of
      // the VBox based on the sum of the heights
      // of the components.
      this.addEventListener(
mx.events.FlexEvent.CREATION_COMPLETE,
                    vboxCompleteHandler);
    }//end constructor
```

There are numerous differences between the code in the constructors for **QuizA** and **QuizB** . Every statement that needs to access a reference pointing to one of the six component objects in Figure 1 is different between the two constructors because of the difference in the way those references are stored. There are other differences as well, which are shown in Listing 6.

Constructor for QuizB

The constructor for the class named **QuizB** is shown in Listing 7.

Example:
Constructor for QuizB.

```
    public function QuizB(){//constructor
      width=vboxWidth;
      setStyle("borderStyle","solid");
      setStyle("backgroundColor",0xffff00);

      components[0].editable = false;//theQuestion
      components[0].width=vboxWidth - 2;

      components[1].groupName="radioButtonGroup";
      components[2].groupName="radioButtonGroup";
      components[3].groupName="radioButtonGroup";

      //checkButton
      components[4].label = "Click to Check
```

```
Answer";

components[4].addEventListener(MouseEvent.CLICK,

checkButtonHandler);

      //result
      components[5].editable = false;
      components[5].width=vboxWidth - 2;
      components[5].visible=false;

      //Add GUI components to the VBox.
      for(var cnt:int = 0;cnt <
components.length;cnt++){
        addChild(components[cnt]);
      }//end for loop

      //Register an event listener that will be
      // executed when this VBox object has been
fully
      // constructed. It will set the height of
      // the VBox based on the sum of the heights
      // of the components.
      this.addEventListener(

mx.events.FlexEvent.CREATION_COMPLETE,
                    vboxCompleteHandler);
    }//end constructor
```

In addition to the differences in the way that references to the six component objects are accessed, Listing 7 contains other significant differences as well.

No code to instantiate the six component objects

First, there is no code in Listing 7 to instantiate the six component objects. As I mentioned earlier, those objects were instantiated and used to populate the six-element array referred to by **components** when the array was created.

Only one statement calling the addChild method in QuizB

Next, you will notice that there are six statement making calls to the **addChild** method of the **VBox** container in Listing 6. Calls to that method cause the components to be added as children of the container.

Those six calls to the **addChild** method were consolidated into a single call inside a **for** loop in Listing 7. This is possible because the six references are contained in an array whose elements can be accessed using a numeric index.

The checkButtonHandler for QuizB

Listing 8 shows the **checkButtonHandler** method for the class named **QuizA** .

Example:
The checkButtonHandler for QuizA.

```
    private function checkButtonHandler(

event:MouseEvent):void{
      result.visible=true;

      if(theAnswer == "0"){
        correctAnswer = choice00.label;
      }else if(theAnswer == "1"){
        correctAnswer = choice01.label;
      }else{
        correctAnswer = choice02.label;
      }//end else
      if((theAnswer=="0" && choice00.selected) ||
           (theAnswer=="1" && choice01.selected)
||
           (theAnswer=="2" && choice02.selected)){

        result.setStyle("color",0x00ff00);
        result.text = "Correct\nCorrect Answer is:
"
                     + correctAnswer;
      }else{
        result.setStyle("color",0xff0000);
        result.text = "Wrong\nCorrect Answer is: "
                     + correctAnswer;
      }//end else
    }//end checkButtonHandler
```

This is the event handler method that is registered for a **click** event on the **Button** by the code in Listing 6.

The checkButtonHandler for QuizB

Listing 9 shows the corresponding **checkButtonHandler** method for the class named **QuizB** .

Example:
The checkButtonHandler for QuizB.

```
    private function checkButtonHandler(

event:MouseEvent):void{
      components[5].visible=true;
      if(theAnswer == "0"){
        correctAnswer = components[1].label;
      }else if(theAnswer == "1"){
        correctAnswer = components[2].label;
      }else{
        correctAnswer = components[3].label;
      }//end else

      if((theAnswer=="0" &&
components[1].selected) ||
         (theAnswer=="1" &&
components[2].selected) ||
         (theAnswer=="2" &&
components[3].selected)){

        components[5].setStyle("color",0x00ff00);
        components[5].text =
                           "Correct\nCorrect
Answer is: "
                           + correctAnswer;
      }else{
        components[5].setStyle("color",0xff0000);
        components[5].text = "Wrong\nCorrect
Answer is: "
                     + correctAnswer;
      }//end else
    }//end checkButtonHandler
```

This is the event handler method that is registered for a **click** event on the **Button** by the code in Listing 7.

Differences between the code

The differences between the methods named **checkButtonHandler** in the two classes result from the different access requirements for the three radio buttons and the text area at the bottom of the question objects in Figure 1.

In one case *(QuizA)* , access is by way of the named reference variables that were declared in Listing 2. In the other case *(QuizB)* , access to each component object's reference is by way of an element of the array that was created in Listing 3.

The vboxCompleteHandler for QuizA

The **vboxCompleteHandler** method for the class named **QuizA** is shown in Listing 10.

Example:
The vboxCompleteHandler for QuizA.

```
    private function vboxCompleteHandler(

event:mx.events.FlexEvent):void{

      this.height =
        theQuestion.height
        + choice00.height
        + choice01.height
        + choice02.height
        + checkButton.height
        + result.height
        + 36;//six spaces per compnent
    }//end vboxCompleteHandler

//==================================================
==//
  }//end class
}//end package
```

Registered by the code in Listing 6

This is the event handler method that was registered on the **VBox** container by the code near the bottom of Listing 6. The purpose of this event handler is to execute when the **VBox** construction is complete and to set the height of the **VBox** container to the heights of the six individual components plus six pixels per component to account for the space between components.

Listing 10 accesses the individual height values by way of the six reference variables declared in Listing 2.

The vboxCompleteHandler for QuizB

The **vboxCompleteHandler** method for **QuizB** is shown in Listing 11.

Example:
The vboxCompleteHandler for QuizB.

```
    private function vboxCompleteHandler(

event:mx.events.FlexEvent):void{

      this.height = 0;

      for(var cnt:int = 0;cnt <
components.length;cnt++){
        this.height += components[cnt].height + 6;
      }//end for loop

    }//end vboxCompleteHandler

//==================================================
==//
  }//end class
}//end package
```

Same purpose as before

This event handler method has the same purpose as the event handler method with the same name in Listing 10.

Once again, because the references to the components are stored in an array, a **for** loop can be used to access and get the height of each of the components and to compute the overall height as the sum of those heights plus six pixels for each component.

The end of the program

Listing 10 and Listing 11 each signal the end of the class and the end of the program.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 12, Listing 13, and Listing 14. Use that code to create a Flex project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the Flex MXML and ActionScript source code discussed in this lesson are provided in Listing 12 through Listing 14.

Example:

Source code for Encapsulation01.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<!--TestGenerator01
  This application illustrates the concept of
exposing
  the interface and hiding the implementation. Two
  classes are defined from which custom components
are
  instantiated. Components instantiated from both
classes
  have the same user interface but they have
radically
  different implementations.-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">
  <mx:Label text="ENCAPSULATION DEMO"/>
  <mx:Label text=" Copyright 2009 R.G.Baldwin" />


<!--The following code instantiates an object of
the class
named QuizA for a multiple-choice quiz question
with three
choices.-->
<cc:QuizA
question=
"Which of the following is not the name of one of
the
seven dwarfs?"
choice0="Dopey"
choice1="Sneezy"
choice2="Harold"
answer="2"
```

```
/>

<!--The following code instantiates an object of
the class
named QuizB for a multiple-choice quiz question
with three
choices. Note that the interface is exactly the
same as
for the class named QuizA. However, the
implementation of
QuizB is radically different from QuizA.-->
<cc:QuizB
question=
"Which of the following is not the name of one of
the
seven dwarfs?"
choice0="Dopey"
choice1="Sneezy"
choice2="Harold"
answer="2"
/>

  <!--The purpose of the follow code is to control
the
      appearance of the GUI components.-->
  <mx:Style>
  RadioButton {
  fontWeight: bold;
  fontSize: 14;
  }
  Label{
  fontWeight: bold;
  fontSize: 18;
  color:  #FFFF00;
  }
  Button{
  fontWeight: bold;
  fontSize: 14;
  }
  TextArea{
  fontWeight: bold;
  fontSize: 14;
  }
  </mx:Style>
</mx:Application>
```

Example:
Source code for QuizA.as.

```
package CustomClasses{
    import flash.events.*;
    import mx.containers.VBox;
    import mx.controls.Button;
    import mx.controls.RadioButton;
    import mx.controls.TextArea;
    import mx.events.FlexEvent;

  public class QuizA extends VBox{
    private var theQuestion:TextArea;
    private var choice00:RadioButton;
    private var choice01:RadioButton;
    private var choice02:RadioButton;
    private var checkButton:Button;
    private var result:TextArea;

    private var theAnswer:String;//numeric string
    private var correctAnswer:String;//actual
string
    private var vboxWidth:int = 375;

//==================================================
==//

    public function set
question(textIn:String):void{
        theQuestion.text = textIn;
    }//end implicit setter

    public function set
answer(answerIn:String):void{
        theAnswer = answerIn;
    }//end implicit setter

    public function set
choice0(choice:String):void{
        choice00.label=choice;
    }//end implicit setter

    public function set
choice1(choice:String):void{
        choice01.label=choice;
    }//end implicit setter

    public function set
choice2(choice:String):void{
        choice02.label=choice;
    }//end implicit setter

//==================================================
==//

    public function QuizA(){//constructor
      width=vboxWidth;
      setStyle("borderStyle","solid");
      setStyle("backgroundColor",0xffff00);

      theQuestion = new TextArea();
      theQuestion.editable = false;
      theQuestion.width=vboxWidth - 2;
      addChild(theQuestion);

      choice00 = new RadioButton();
      choice00.groupName="radioButtonGroup";
      addChild(choice00);

      choice01 = new RadioButton();
      choice01.groupName="radioButtonGroup";
      addChild(choice01);

      choice02 = new RadioButton();
      choice02.groupName="radioButtonGroup";
      addChild(choice02);

      checkButton = new Button();
      checkButton.label = "Click to Check Answer";

checkButton.addEventListener(MouseEvent.CLICK,

checkButtonHandler);
      addChild(checkButton);

      result = new TextArea();
      result.editable = false;
      result.width=vboxWidth - 2;
      result.visible=false;
      addChild(result);

      //Register an event listener that will be
      // executed when this object has been fully
      // constructed. It will set the height of
      // the VBox based on the sum of the heights
      // of the components.
      this.addEventListener(

mx.events.FlexEvent.CREATION_COMPLETE,
                    vboxCompleteHandler);
    }//end constructor

//==================================================
==//

    private function checkButtonHandler(

event:MouseEvent):void{
      result.visible=true;

      if(theAnswer == "0"){
        correctAnswer = choice00.label;
      }else if(theAnswer == "1"){
        correctAnswer = choice01.label;
      }else{
        correctAnswer = choice02.label;
      }//end else

      if((theAnswer=="0" && choice00.selected) ||
           (theAnswer=="1" && choice01.selected)
||
           (theAnswer=="2" && choice02.selected)){

        result.setStyle("color",0x00ff00);
        result.text = "Correct\nCorrect Answer is:
"
                     + correctAnswer;
      }else{
        result.setStyle("color",0xff0000);
        result.text = "Wrong\nCorrect Answer is: "
                     + correctAnswer;
      }//end else
    }//end checkButtonHandler

//==================================================
==//
```

```
    private function vboxCompleteHandler(

event:mx.events.FlexEvent):void{
      //Set the height equal to the sum of the
      // heights of the components plus six
      // pixels per component to account for the
      // space between components.
      this.height =
        theQuestion.height
        + choice00.height
        + choice01.height
        + choice02.height
        + checkButton.height
        + result.height
        + 36;//six spaces per compnent
    }//end vboxCompleteHandler

//==================================================
==//
  }//end class
}//end package
```

Example:
Source code for QuizB.as.

```
//This is an update of the class named QuizA. This
version
// stores references to all of the GUI components
in a
// six-element array and uses for loops to process
them
// where appropriate. Note that the GUI components
are
```

```
// instantiated and their references are stored in
the
// array when the array is created.

package CustomClasses{
    import flash.events.*;
    import mx.containers.VBox;
    import mx.controls.Button;
    import mx.controls.RadioButton;
    import mx.controls.TextArea;
    import mx.events.FlexEvent;

  public class QuizB extends VBox{

    //References to six GUI components are stored
in the
    // following array.
    private var components:Array =
        new Array(new TextArea(),//theQuestion
                  new RadioButton(),
                  new RadioButton(),
                  new RadioButton(),
                  new Button(),//checkButton
                  new TextArea());//result

    private var theAnswer:String;//numeric string
    private var correctAnswer:String;//actual
string
    private var vboxWidth:int = 375;

//==================================================
==//

    public function set
question(textIn:String):void{
        components[0].text = textIn;
    }//end implicit setter
    public function set
answer(answerIn:String):void{
        theAnswer = answerIn;
    }//end implicit setter

    public function set
choice0(choice:String):void{
        components[1].label=choice;
    }//end implicit setter

    public function set
choice1(choice:String):void{
        components[2].label=choice;
    }//end implicit setter

    public function set
choice2(choice:String):void{
        components[3].label=choice;
    }//end implicit setter

//==================================================
==//

    public function QuizB(){//constructor
      width=vboxWidth;
      setStyle("borderStyle","solid");
      setStyle("backgroundColor",0xffff00);

      components[0].editable = false;//theQuestion
      components[0].width=vboxWidth - 2;

      components[1].groupName="radioButtonGroup";
      components[2].groupName="radioButtonGroup";
      components[3].groupName="radioButtonGroup";

      //checkButton
      components[4].label = "Click to Check
Answer";

components[4].addEventListener(MouseEvent.CLICK,

checkButtonHandler);

      //result
      components[5].editable = false;
      components[5].width=vboxWidth - 2;
      components[5].visible=false;

      //Add GUI components to the VBox.
      for(var cnt:int = 0;cnt <
components.length;cnt++){
        addChild(components[cnt]);
      }//end for loop

      //Register an event listener that will be
      // executed when this VBox object has been
fully
      // constructed. It will set the height of
      // the VBox based on the sum of the heights
      // of the components.
      this.addEventListener(

mx.events.FlexEvent.CREATION_COMPLETE,
                    vboxCompleteHandler);
    }//end constructor

//==================================================
==//

    private function checkButtonHandler(

event:MouseEvent):void{
      components[5].visible=true;
      if(theAnswer == "0"){
        correctAnswer = components[1].label;
      }else if(theAnswer == "1"){
        correctAnswer = components[2].label;
      }else{
        correctAnswer = components[3].label;
      }//end else

      if((theAnswer=="0" &&
components[1].selected) ||
         (theAnswer=="1" &&
components[2].selected) ||
         (theAnswer=="2" &&
components[3].selected)){

        components[5].setStyle("color",0x00ff00);
        components[5].text =
                           "Correct\nCorrect
Answer is: "
                           + correctAnswer;
      }else{
        components[5].setStyle("color",0xff0000);
        components[5].text = "Wrong\nCorrect
Answer is: "
                     + correctAnswer;
      }//end else
    }//end checkButtonHandler

//==================================================
==//

    private function vboxCompleteHandler(

event:mx.events.FlexEvent):void{
      //Set the height equal to the sum of the
      // heights of the components plus six
      // pixels per component to account for the
      // space between components.
      this.height = 0;
      for(var cnt:int = 0;cnt <
components.length;cnt++){
        this.height += components[cnt].height + 6;
      }//end for loop

    }//end vboxCompleteHandler

//==================================================
==//
  }//end class
}//end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Encapsulation - The Big Picture
- Files:
  - ActionScript0106\ActionScript0106.htm
  - ActionScript0106\Connexions\ActionScriptXhtml0106.htm

you should be aware that some of the HTML elements in this module may not translate well into PDF.

-end-

Inheritance - The Big Picture
Learn how to use inheritance to create custom components. Also learn how to use ActionScript skins to change the appearance of a standard component.

Note: Click Skins01 to run this ActionScript program. *(Click the "Back" button in your browser to return to this page.)*

Table of contents

Preface

General

Note: Note that all references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a continuing series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

The three main characteristics of an object-oriented program

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

I explained encapsulation from a big-picture viewpoint in the previous lesson. *(See Baldwin's ActionScript programming website .)* In this lesson and the next, I will explain and illustrate inheritance and polymorphism from a big-picture viewpoint. Following that, I will get down in the weeds and start explaining in detail how to use ActionScript for object-oriented programming *(OOP)* .

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Most of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

The first major characteristic of an object-oriented program is encapsulation, which I explained in the previous lesson. The second of the three major characteristics is *inheritance* , followed by *polymorphism* . I will explain inheritance from a big-picture viewpoint in this lesson and will explain polymorphism in the next lesson.

A new class can *extend* an existing class

A class can be defined to inherit the properties, events, and methods of another class. From a syntax viewpoint, this is accomplished using the **extends** keyword.

The class being extended is often called the *base class* or the *superclass* , and the new class is often called the *derived class* or the *subclass* .

What is inherited?

The subclass inherits the data representation and behavior of the superclass *(and all of its superclasses)* . However, the subclass can modify the behavior of inherited methods by overriding them. *(That will be one of the topics in the next lesson on polymorphism.)* The subclass can also add new data representation and behavior that is unique to its own purposes.

The superclass remains viable

A program can instantiate objects of a superclass as well as from its subclass. From a practical viewpoint, the superclass doesn't even know that it has been extended.

A hierarchy of classes

Inheritance is hierarchical. In other words, a class may be the subclass of one *(and only one)* other class and may be the superclass of one or more other classes.

For example, the ActionScript 3 **Button** class is a subclass of the **UIComponent** class, and is the superclass of the following classes:

- AccordianHeader
- CheckBox
- LinkButton
- PopUpButton
- RadioButton
- ScrollThumb
- SliderThumb

An orderly hierarchy

Inheritance mechanisms allow you build an orderly hierarchy of classes.

When several of your abstract data types have characteristics in common, you can design their commonalities into a single superclass and separate their unique characteristics into unique subclasses. This is one of the purposes of inheritance. *(This purpose is illustrated by the program that I will explain later in this lesson.)*

The airship hierarchy

For example, suppose you are building a program dealing with airships. All airships have altitude and range parameters in common. Therefore, you could build a base **Airship** class containing data and methods having to do with range and altitude.

From this superclass, you might derive a **Balloon** class and an **Airplane** class.

The Balloon class

The **Balloon** class might add variables and methods dealing with passenger capacity and what makes it go up *(helium, hydrogen, or hot air)* . Objects of the **Balloon** class would then be able to deal with altitude, range, passenger capacity, and what makes it go up.

The Airplane class

The **Airplane** class might add variables and methods dealing with engine type *(jet or propeller)* and cargo capacity. Objects of the **Airplane** class could then deal with altitude, range, engine type, and cargo capacity.

Three types of objects

Having created this hierarchy of classes, you could instantiate objects of type **Airship** , **Balloon** , and **Airplane** with the objects of each type having variables and methods to deal with those special characteristics of the flying machine indicated by the name of the class.

From the general to the specialized

You may have noticed that in this hierarchical class structure, inheritance causes the structure to grow in a direction from most general to more specialized. This is typical.

Single and multiple inheritance

C++ and some other object-oriented programming languages allow for multiple inheritance. This means that a new class can extend more than one superclass. This has advantages in some cases, but can lead to difficulties in other cases.

ActionScript 3 does not support multiple inheritance. Instead it supports a different mechanism called an *interface* that provides most of the benefits of multiple inheritance without most of the problems. I will explain the ActionScript interface in a future lesson.

The ISA relationship

You will sometimes hear people speak of the **ISA** relationship when discussing OOP. The source of this terminology is more fundamental than you may at first suspect.

Object-oriented designers often strive to use inheritance to model relationships where a subclass *"is a kind of"* the superclass. For example, a car *"is a kind of"* vehicle. A programmer *"is a kind of"* employee which in turn *"is a kind of"* person.

This relationship is called the **ISA** relationship. It's that simple.

Preview

Assume that you are working on children's games. You probably don't want to use buttons that cause your game program to look like a business program. Instead, you might prefer to use bright-colored buttons that look like animals, flowers, or other child-pleasing objects such as butterflies and frogs. *(See Figure 1 for example.)*

Characteristics in common

On the other hand, you want your buttons to exhibit all of the functionality of the standard ActionScript **Button** class. In other words, you want your new buttons to have many characteristics in common with the ActionScript **Button** class.

Note: A new class may not be required. -- It is probably also possible to accomplish this using embedded ActionScript code, but I will go the class definition route to illustrate inheritance and to produce a "cleaner" solution.

Inheritance to the rescue

As it turns out, this is easy to accomplish by defining an ActionScript class for each of your new button types.

The folks who defined the **Button** class have already designed the desired characteristics into the **Button** class and its superclasses. We can use inheritance to take advantage of the existing characteristics and to add new characteristics by defining unique subclasses of the **Button** class.

Three new button classes

The program named **Skins01** that I will explain in this lesson defines three new button classes named **ButterflyButton** , **FrogButton** , and **FancyButton** . Each of these classes extends the **Button** class and inherits all of the characteristics of a standard **Button** object. However, the visual elements of the standard button are modified in the subclasses through the use of the ActionScript *skinning* capability.

What is skinning?

According to *About skinning* ,

> *"Skinning is the process of changing the appearance of a component by modifying or replacing its visual elements. These elements can be made up of bitmap images, SWF files, or class files that contain drawing methods that define vector images.*
>
> *Skins can define the entire appearance, or only a part of the appearance, of a component in various states. For example, a Button control has eight possible states, and eight associated skin properties."*

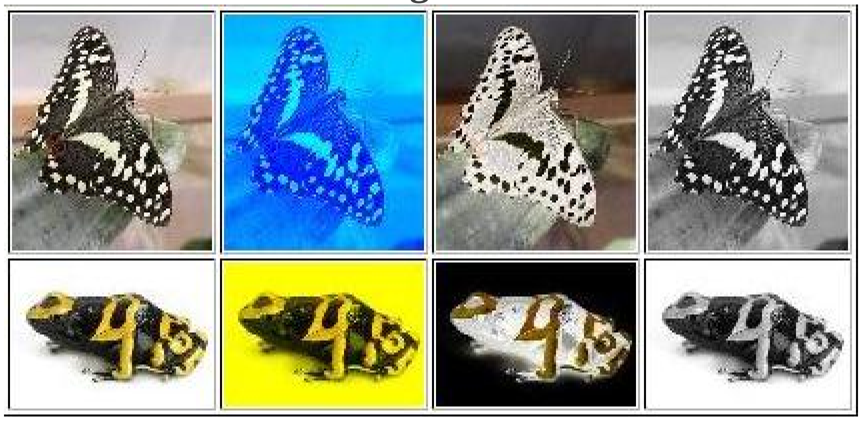
Will extend the Button class

In this lesson, I will extend the **Button** class into three new classes and will modify four of the eight skin properties for objects instantiated from the three new classes. I will also define a fourth class named **Driver** that is designed to exercise and show how to use the new buttons.

Images used as skins for the buttons

Figure 1 shows the images used to define new skins for the three buttons.
Images used as skins for the buttons.

Figure



Images used as skins for the buttons.

The button class named **ButterflyButton** was skinned with the four butterfly images in the top row. The button class named **FrogButton** was skinned with the four frog images in the bottom row. The button class named **FancyButton** was skinned with both butterfly images and frog images and was given the ability to toggle back and forth between butterflies and frogs.

The toggleSkin method

The **FancyButton** class exposes a public method named **toggleSkin** . The rightmost button in Figure 2 is an object of the **FancyButton** class. Whenever this button is clicked, a **click** event handler registered on the button calls the **toggleSkin** method. Each time this method is called, the button's skins toggle from butterfly to frog or vice versa. *(See the rightmost button in Figure 5.)* This illustrates the capability to change skins on an ActionScript component at runtime.

Note: Spacebar -- Even though this text concentrates on the manipulation of the buttons with the mouse, a button can also be manipulated with the space bar when it has the focus. Note, however, that this lesson doesn't address the topic of focus.

Skins versus button styles

The different skins on the buttons are exposed by associating the images in Figure 1 with various button styles.

The *upSkin style*

The two images in the first column in Figure 1 show what the buttons look like when the mouse is not hovering over or pointing at the button. This corresponds to the button style named *"upSkin"* ,

The *overSkin style*

The two images in the second column show what the buttons look like when the mouse pointer is hovering over the button but the left mouse button has not been pressed. This corresponds to the button style named *"overSkin"* .

The *downSkin style*

The two images in the third column show what the buttons look like when the mouse pointer is over the button and the left mouse button is pressed. This corresponds to the button style named *"downSkin"* .

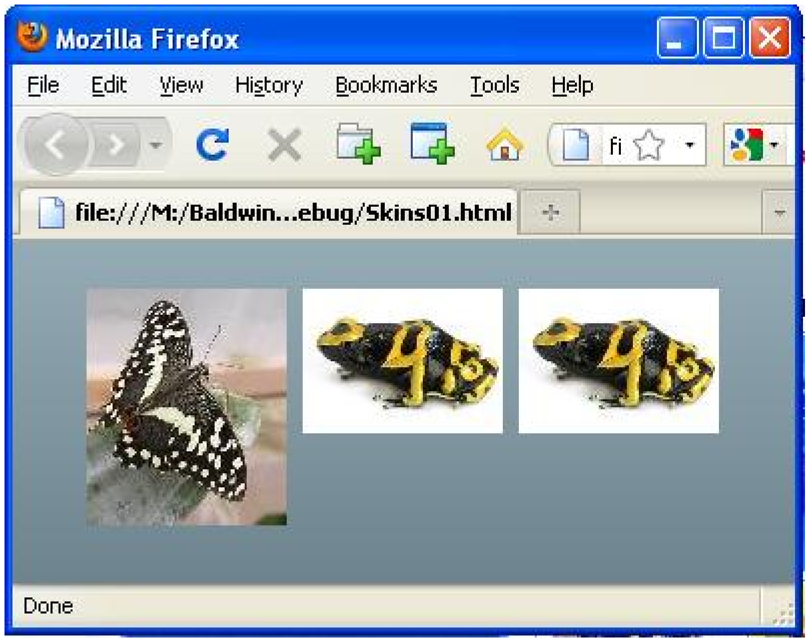
The *disabledSkin style*

The two images in the fourth column show what the buttons look like when the enabled property of the button is set to false. This corresponds to the button style named *"disabledSkin"* . Note, however, that this program does not expose the disabled state of the buttons.

Program output at startup

Figure 2 shows the three buttons at startup.
Program output at startup.

Figure


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///M:/Baldwin...ebug/Skins01.html
Done

Program output at startup.

Button skins at startup

As you can see, all three buttons show the images from the first column in Figure 1 at startup.

An object of the ButterflyButton class

The leftmost button in Figure 2, which is an object of the **ButterflyButton** class, always shows one of the images in the top row of Figure 1, depending on the interaction with the mouse.

An object of the FrogButton class

The middle button in Figure 2, which is an object of the **FrogButton** class, always shows one of the images in the bottom row of Figure 1 depending on the interaction with the mouse.

An object of the FancyButton class

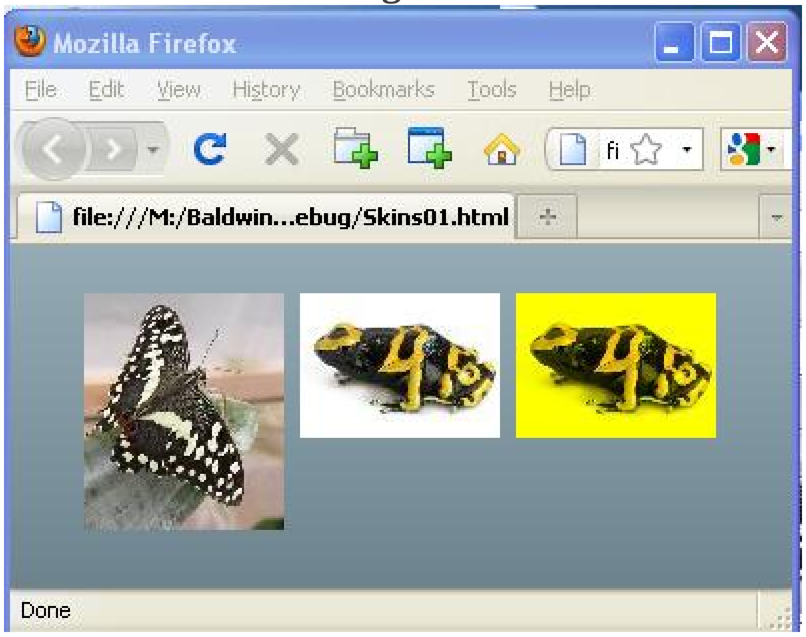
The rightmost button in Figure 2 is an object of the **FancyButton** class. As mentioned earlier, the object of the **Driver** class registers a **click** event handler this button. Whenever the rightmost button is clicked with the left mouse button, the **toggleSkin** method belonging to the rightmost button is called causing the skin images to toggle between frogs and butterflies.

At startup, this button shows the frog image from the first column in Figure 1. After it has been toggled, it shows the butterfly image from the first column in Figure 1 as the **upSkin** style.

Mouse pointer hovering over the rightmost button

Figure 3 shows the result of allowing the mouse pointer to hover over the rightmost button without pressing the left mouse button.
Mouse pointer hovering over the rightmost button.

Figure


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///M:/Baldwin...ebug/Skins01.html
Done

Mouse pointer hovering over the rightmost button.

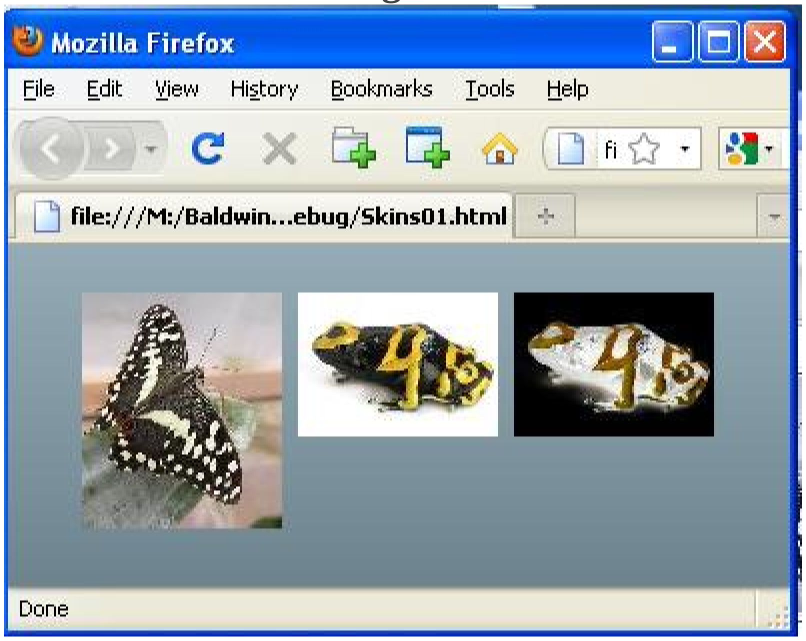
This action causes the button to switch to the **overSkin** style, which in turn causes one of the two images in the second column in Figure 1 to be exposed.

Mouse pointer pressed on the rightmost button

Figure 4 shows the result of pointing to the rightmost button with the mouse pointer and pressing the left mouse button.

Mouse pointer pressed on the rightmost button.

Figure


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///M:/Baldwin...ebug/Skins01.html
Done

Mouse pointer pressed on the rightmost button.

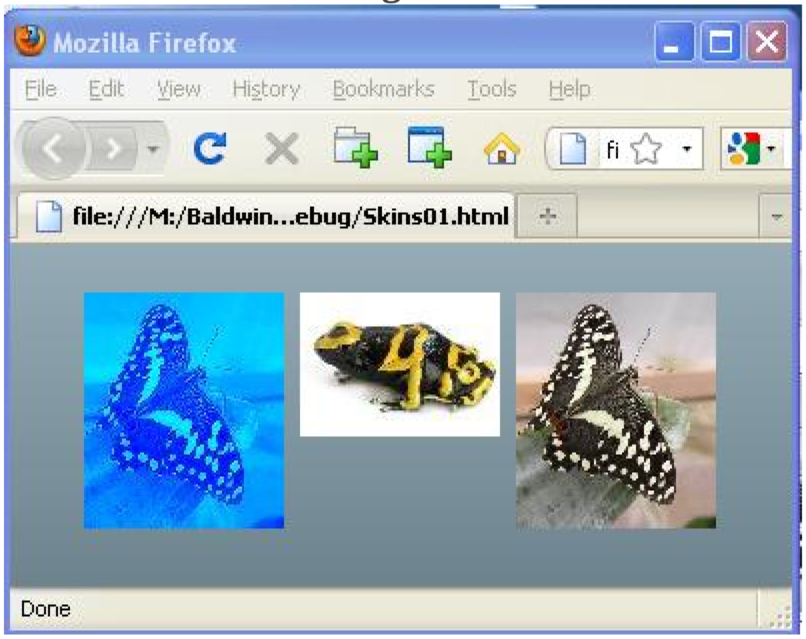
Note: A negative image -- In case you are interested, the images in the third column in Figure 1 are the negative of the images in the first column.

This causes the button to switch to the **downSkin** style, which in turn causes one of the images from the third column in Figure 1 to be exposed.

Mouse pointer hovering over the leftmost button

Figure 5 shows the result of allowing the mouse pointer to hover over the leftmost button.
Mouse pointer hovering over the leftmost button.

Figure


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///M:/Baldwin...ebug/Skins01.html
Done

Mouse pointer hovering over the leftmost button.

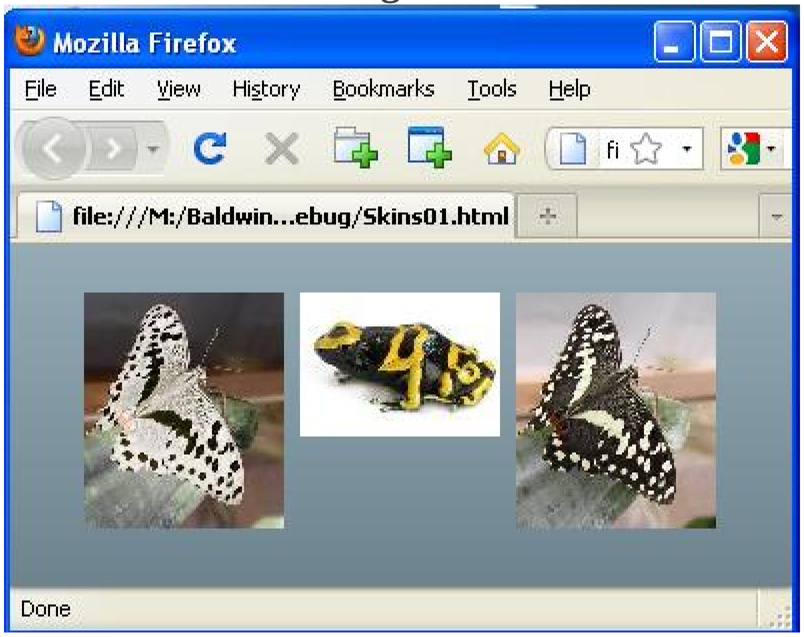
Hovering over the button with the mouse pointer causes the button to switch to the **overSkin** style exposing the top image in the second column in Figure 1.

Note that the rightmost button has been toggled from frog skins to butterfly skins in Figure 5.

Mouse pointer pressed on the leftmost button

Figure 6 shows the result of pointing to the leftmost button and pressing the left mouse button.
Mouse pointer pressed on the leftmost button.

Figure


Mozilla Firefox
File Edit View History Bookmarks Tools Help
file:///M:/Baldwin...ebug/Skins01.html
Done

Mouse pointer pressed on the leftmost button.

This causes the button to switch to the **downSkin** style exposing the top image in the third column in Figure 1.

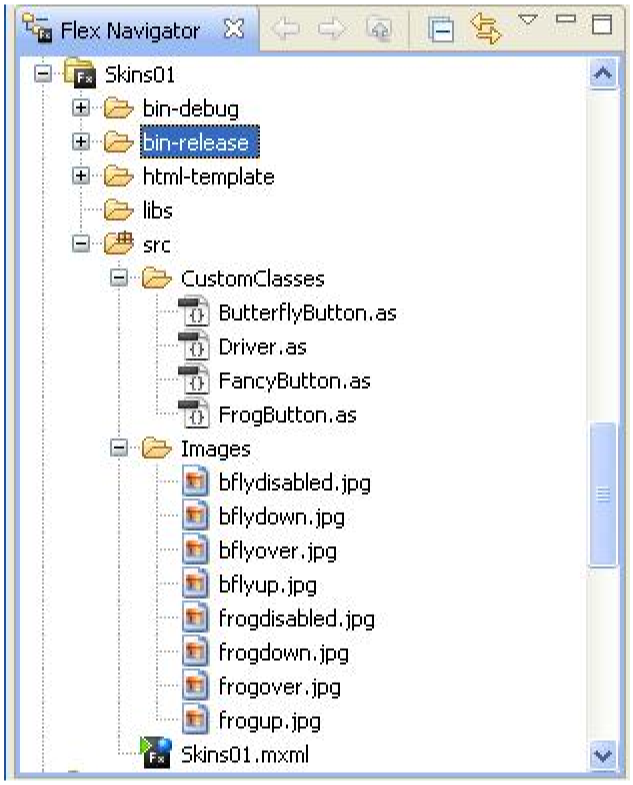
Discussion and sample code

Skins01 project file structure

Figure 7 shows the project file structure for the Flex project named **Skins01.**

Skins01 project file structure.

Figure



Skins01
bin-debug
bin-release
html-template
libs
src
CustomClasses
ButterflyButton.as
Driver.as
FancyButton.as
FrogButton.as
Images
bflydisabled.jpg
bflydown.jpg
bflyover.jpg
bflyup.jpg
frogdisabled.jpg
frogdown.jpg
frogover.jpg
frogup.jpg
Skins01.mxml

Skins01 project file structure.

The image in Figure 7 was captured from the Flex Navigator panel in the Flex Builder 3 IDE.

Eight images

The eight images shown in Figure 1 are listed in the **Images** folder in Figure 7.

Three custom button classes

The three custom button classes and the driver class illustrated by Figure 2 through Figure 6 are listed in the **CustomClasses** folder in Figure 7.

The MXML application file

And of course, the Flex MXML file is shown as **Skins01.mxml** in Figure 7.

Custom button classes are very similar

The three custom button classes are very similar. The class named **FancyButton** is the most complex of the three. Therefore, I will explain the class named **FancyButton** in detail but I won't explain the other two custom button classes.

In addition, I will explain the MXML file named **Skins01.mxml** and the class file named **Driver.as** .

Will explain in fragments

I will explain the code in fragments. Complete listings of all of the source code files are provided in Listing 8 through Listing 12 near the end of the lesson.

The file named Skins01.mxml

A complete listing of this file is shown in Listing 1. In addition, a complete listing is also provided in Listing 8 near the end of the lesson along with the code for all of the other files in this application.

Example:
Complete listing for Skins01.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

The code in Listing 1 couldn't be simpler. It declares a namespace for the folder containing the four class files in Figure 7 and instantiates an object of the class named **Driver** . In this case, the Flex application is simply serving as a launch pad for the ActionScript program.

The file named Driver.as

The class named **Driver** begins in Listing 2. A complete listing of this class file is provided in Listing 9 near the end of the lesson.

Example:
Beginning of the class named Driver.

```
package CustomClasses{
  import flash.events.*;
  import mx.containers.HBox;

  public class Driver extends HBox{

    private var bFlyButton:ButterflyButton =
                                   new
ButterflyButton();
    private var frogButton:FrogButton = new
FrogButton();
    private var fancyButton:FancyButton =
                                       new
FancyButton();
```

Purpose of the class

The purpose of this class is to define the characteristics of an object that will serve as a driver to exercise and to illustrate the behavior of objects of the following custom classes:

- **ButterflyButton**
- **FrogButton**
- **FancyButton**

Inheritance in action

Note that this class extends the standard Flex/ActionScript class named **HBox** . Thus, an object of the **Driver** class inherits all of the characteristics defined into the **HBox** class and adds more characteristics that are unique to an object of the **Driver** class.

For example, one of the important characteristics inherited from the **HBox** class is the ability to arrange child components horizontally as shown in Figure 2.

Instantiate custom button objects

Listing 2 instantiates objects of the three custom classes listed above and stores references to those objects in private instance variables with the following names:

- **bFlyButton**
- **frogButton**
- **fancyButton**

Storing the references as private instance variables makes them accessible to the constructor and to methods defined within the **Driver** class, but makes them inaccessible otherwise.

The constructor for the Driver class

Listing 3 defines the constructor for the **Driver** class.

Example:
The constructor for the Driver class.

```
    public function Driver(){//constructor
      addChild(bFlyButton);
      addChild(frogButton);
      addChild(fancyButton);

      fancyButton.addEventListener(

MouseEvent.CLICK,buttonHandler);
    }//end constructor
```

Listing 3 begins by adding the three objects instantiated in Listing 2 to the object being constructed, which is possible because an object of the **Driver** class is an **HBox** container. *(See the earlier section titled The ISA relationship .)*

Then Listing 3 registers a **click** event handler method named **buttonHandler** on the **FancyButton** object.

The method named buttonHandler

The click event handler method named **buttonHandler** is shown in Listing 4.

Example:
The method named buttonHandler .

```
    private function buttonHandler(

event:MouseEvent):void{
      fancyButton.toggleSkin();
    }//end buttonHandler
```

```
  }//end class
}//end package
```

This method is executed each time the **FancyButton** object fires a **click** event. When the method is executed, it calls the **toggleSkin** method on the **FancyButton** object. As you will see later, the **toggleSkin** method causes the **FancyButton** object to toggle its skins between butterfly images and frog images.

The end of the Driver class

Listing 4 also signals the end of the class named **Driver** .

The file named FancyButton.as

The **FancyButton** class begins in Listing 5. A complete listing of the class file is provided in Listing 10 near the end of the lesson.

Define a custom button using skins

This class defines a custom button using skins. When the program starts, frog images are used to define the button's skins as shown by the rightmost button in Figure 2. Each time the **toggleSkin** method is called, the skin switches between frog images and butterfly images.

Example:
Beginning of the FancyButton class.

```
package CustomClasses{
  import mx.controls.Button;

  public class FancyButton extends Button{
    [Embed("/Images/frogup.jpg")]
    private var frogUp:Class;

    [Embed("/Images/frogover.jpg")]
    private var frogOver:Class;

    [Embed("/Images/frogdown.jpg")]
    private var frogDown:Class;

    [Embed("/Images/frogdisabled.jpg")]
    private var frogDisabled:Class;

    [Embed("/Images/bflyup.jpg")]
    private var bFlyUp:Class;

    [Embed("/Images/bflyover.jpg")]
    private var bFlyOver:Class;

    [Embed("/Images/bflydown.jpg")]
    private var bFlyDown:Class;

    [Embed("/Images/bflydisabled.jpg")]
    private var bFlyDisabled:Class;
```

A FancyButton object ISA Button object

The **FancyButton** class extends the **Button** class. Therefore, an object of the **FancyButton** class is also a **Button** object, possessing all of the characteristics of a **Button** . *(See the earlier section titled The ISA relationship .)* One of the characteristics of a **Button** object is the ability to have its appearance changed through the use of skinning.

Different ways to create skins

There are several different ways to create skins for ActionScript objects, including the use of images to create *graphical skins* . According to About Skinning ,

> *"When using graphical skins, you must embed the image file for the skin in your application. To specify your skin, you can use the setStyle() method, set it inline, or use Cascading Style Sheets (CSS)"*

Embedding image files

There is more than one way to embed an image file in your application and I won't attempt to explain the details. *(I suggest that you go to Google and search for* ***ActionScript embed metadata tag*** *for more information.)*

The embedding syntax

The syntax that you see in Listing 5 *"Imports JPEG, GIF, PNG, SVG, and SWF files at compile time."*

The syntax also makes those images accessible by way of the variables that are declared immediately following each **Embed** tag. Therefore, the images from each of the eight image files imported in Listing 5 can be referenced later using the eight variables declared in Listing 5.

Constructor for the FancyButton class

The constructor for the **FancyButton** class is shown in Listing 6.

Example:
Constructor for the FancyButton class.

```
    public function FancyButton(){//constructor

      this.setStyle("upSkin", frogUp);
      this.setStyle("overSkin", frogOver);
      this.setStyle("downSkin", frogDown);
      this.setStyle("disabledSkin", frogDisabled);

    }//end constructor
```

The constructor calls the **setStyle** method four times in succession to set the four styles described earlier to the frog images imported in Listing 5. This causes the button instantiated from the **FancyButton** class to exhibit frog skins at startup as shown in Figure 2.

The toggleSkin method

The **toggleSkin** method is shown in Listing 7.

Example:
The toggleSkin method.

```
    public function toggleSkin():void{
      if(this.getStyle("upSkin") == frogUp){
        this.setStyle("upSkin", bFlyUp);
        this.setStyle("overSkin", bFlyOver);
        this.setStyle("downSkin", bFlyDown);
        this.setStyle("disabledSkin",
bFlyDisabled);
      }else{
        this.setStyle("upSkin", frogUp);
        this.setStyle("overSkin", frogOver);
        this.setStyle("downSkin", frogDown);
        this.setStyle("disabledSkin",
frogDisabled);
      }//end else

    }//end toggleSkin
  }//end class
}//end package
```

Called by a click event handler method

This method is called whenever the **FancyButton** object that was instantiated in Listing 2 fires a **click** event. This call results from the registration of the event handler method on the button in Listing 3 and the definition of the event handler method in Listing 4.

Skin toggling algorithm

Listing 7 tests to determine if the button is currently exposing skins based on the frog images. If so, it uses the **setStyle** method to switch all four skin styles to the butterfly images.

If not, meaning that it is currently exposing butterfly images, it uses the **setStyle** method to switch all four skin styles to the frog images.

Run the program and see for yourself

If you run this program and click repeatedly on the rightmost button in Figure 2, you will see the skins for the button toggle between frog images and butterfly images. This demonstrates that ActionScript skins can be changed at runtime.

The end of the FancyButton class

Listing 7 also signals the end of the **FancyButton** class. As I mentioned earlier, you will find the complete source code for this class in Listing 10 near the end of the lesson. You will also find the source code for the somewhat simpler classes named **ButterflyButton** and **FrogButton** in Listing 11 and Listing 12.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 8 through Listing 12. Use that code, along with some image files of your own to create a Flex project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the source code for the files used in this Flex application are provided in Listing 8 through Listing 12 below.

Example:
Source code for the file named Skins01.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<!--The purpose of this application is twofold:
    1.  To illustrate the use of inheritance to
create
    custom components.
    2.  To illustrate the use of skins to change
the
    appearance of the standard ActionScript
    components.

    Four custom classes are defined. One uses
butterfly
    images to create a ButterflyButton. Another
uses
    images of a frog to create a FrogButton. A
third
    uses both butterfly and frog images to toggle
the
    skins between the two at runtime.

    The fourth class is a driver class that is
designed
    to exercise the other three classes and to
    demonstrate their use.

    All three classes define the following styles
but the
    disabledSkin style is not illustrated by the
code.

    upSkin
    overSkin
    downSkin
    disabledSkin

    Mxml code is used to instantiate an object of
the
    Driver class.
     -->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>
```

```
</mx:Application>
```

Example:
Source code for the class named Driver.

```
//The purpose of this class is to serve as a
driver to
// exercise the following custom classes:
// ButterflyButton
// FrogButton
// FancyButton
//A click event handler is registered on an object
of the
// FancyButton class.  Each time the object is
clicked
// with the mouse, the toggleSkin method is called
on
// the object causing it to toggle its skin
between
// butterfly images and frog images.


package CustomClasses{
  import flash.events.*;
  import mx.containers.HBox;

  public class Driver extends HBox{

    private var bFlyButton:ButterflyButton =
                                   new
ButterflyButton();
    private var frogButton:FrogButton = new
FrogButton();
```

```
    private var fancyButton:FancyButton =
                                       new
FancyButton();

    public function Driver(){//constructor
      addChild(bFlyButton);
      addChild(frogButton);
      addChild(fancyButton);

      fancyButton.addEventListener(

MouseEvent.CLICK,buttonHandler);
    }//end constructor

    private function buttonHandler(

event:MouseEvent):void{
      fancyButton.toggleSkin();
    }//end buttonHandler

  }//end class
}//end package
```

Example:
Source code for the class named FancyButton.

```
//This class defines a custom button using skins.
When
// the program starts, frog images are used to
define
// the button's skin. Each time the toggleSkin
method is
// called, the skin switches between frog images
and
```

```
// butterfly images.
package CustomClasses{
  import mx.controls.Button;

  public class FancyButton extends Button{
    [Embed("/Images/frogup.jpg")]
    private var frogUp:Class;

    [Embed("/Images/frogover.jpg")]
    private var frogOver:Class;

    [Embed("/Images/frogdown.jpg")]
    private var frogDown:Class;

    [Embed("/Images/frogdisabled.jpg")]
    private var frogDisabled:Class;

    [Embed("/Images/bflyup.jpg")]
    private var bFlyUp:Class;

    [Embed("/Images/bflyover.jpg")]
    private var bFlyOver:Class;

    [Embed("/Images/bflydown.jpg")]
    private var bFlyDown:Class;

    [Embed("/Images/bflydisabled.jpg")]
    private var bFlyDisabled:Class;


    public function FancyButton(){//constructor

      this.setStyle("upSkin", frogUp);
      this.setStyle("overSkin", frogOver);
      this.setStyle("downSkin", frogDown);
      this.setStyle("disabledSkin", frogDisabled);
    }//end constructor

    public function toggleSkin():void{
      if(this.getStyle("upSkin") == frogUp){
        this.setStyle("upSkin", bFlyUp);
        this.setStyle("overSkin", bFlyOver);
        this.setStyle("downSkin", bFlyDown);
        this.setStyle("disabledSkin",
bFlyDisabled);
      }else{
        this.setStyle("upSkin", frogUp);
        this.setStyle("overSkin", frogOver);
        this.setStyle("downSkin", frogDown);
        this.setStyle("disabledSkin",
frogDisabled);
      }//end else

    }//end toggleSkin
  }//end class
}//end package
```

Example:
Source code for the class named ButterflyButton.

```
//This class defines a custom button using skins.
The
// normal or up position shows an image of a
butterfly.
// When the mouse is over the button, the image
takes on
// a blue tint. When the mouse is pressed on the
button,
// the image is negated. When the button is
disabled,
// the image turns to gray scale.
package CustomClasses{
  import mx.controls.Button;

  public class ButterflyButton extends Button{
    public function ButterflyButton(){

      [Embed("/Images/bflyup.jpg")]
      var up:Class;

      [Embed("/Images/bflyover.jpg")]
      var over:Class;

      [Embed("/Images/bflydown.jpg")]
      var down:Class;

      [Embed("/Images/bflydisabled.jpg")]
      var disabled:Class;

      this.setStyle("upSkin", up);
      this.setStyle("overSkin", over);
      this.setStyle("downSkin", down);
      this.setStyle("disabledSkin", disabled);

      //Can do this to demo the disabled skin.
      //this.enabled=false;

    }//end constructor

  }//end class
}//end package
```

Example:
Source code for the class named FrogButton.

```
//This class defines a custom button using skins.
The
// normal or up position shows an image of a frog.
// When the mouse is over the button, the image
takes on
// a blue tint. When the mouse is pressed on the
button,
// the image is negated. When the button is
disabled,
// the image turns to gray scale.
package CustomClasses{
  import mx.controls.Button;

  public class FrogButton extends Button{
    public function FrogButton(){

      [Embed("/Images/frogup.jpg")]
      var up:Class;

      [Embed("/Images/frogover.jpg")]
      var over:Class;

      [Embed("/Images/frogdown.jpg")]
      var down:Class;

      [Embed("/Images/frogdisabled.jpg")]
      var disabled:Class;

      this.setStyle("upSkin", up);
      this.setStyle("overSkin", over);
      this.setStyle("downSkin", down);
      this.setStyle("disabledSkin", disabled);

      //Can do this to demo the disabled skin.
      //this.enabled=false;

    }//end constructor
  }//end class
}//end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Inheritance - The Big Picture
- Files:
  - ActionScript0108\ActionScript0108.htm
  - ActionScript0108\Connexions\ActionScriptXhtml0108.htm

-end-

Polymorphism - The Big Picture
Learn the essence of runtime polymorphism.

Note: Click Polymorph02 to run this ActionScript program. (Click the "Back" button in your browser to return to this page.)

Table of Contents

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a continuing series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

The three main characteristics of an object-oriented program

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

I explained both encapsulation and inheritance from a big-picture viewpoint in previous lessons. *(See Baldwin's ActionScript programming website .)* There are two different ways to implement polymorphism:

- Polymorphism based on class inheritance
- Polymorphism based on interface inheritance

I will explain and illustrate polymorphism based on class inheritance in this lesson and will explain and illustrate polymorphism based on interface inheritance in the next lesson.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Most of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder

3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

The first of three major characteristics of an object-oriented program is encapsulation, which I explained in a previous lesson. The second of the three major characteristics is inheritance, and the third is polymorphism. I also explained inheritance in a previous lesson. I will explain polymorphism from a big-picture viewpoint in this lesson.

Not as complicated as it sounds

Upon first reading, you may conclude that polymorphism sounds extremely complicated. However, polymorphism is more difficult to explain than it is to program. Once you have read the description and have seen the concept applied to an actual program, you will *(hopefully)* conclude that it is not as complicated as it sounds.

What is polymorphism?

Polymorphism is a word taken from the Greek, meaning "many forms", or words to that effect.

The purpose of polymorphism as it applies to OOP is to allow one name to be used to specify a general class of actions. Within that general class of actions, the specific action that is applied in any particular situation is determined by the type of data involved.

Polymorphism in ActionScript comes into play when inherited methods are overridden to cause them to behave differently for different types of subclass objects.

Overriding versus overloading methods

If you read much in the area of OOP, you will find the words *override* and *overload* used frequently. *(This lesson deals with overriding methods and does not deal with overloading.)*

Some programming languages such as C++, Java, and C# support a concept known as method or constructor overloading. However, ActionScript 3 does not support method or constructor overloading. Overriding a method is an entirely different thing from overloading a method even for those languages that support overloading.

Modify the behavior of an inherited method

Polymorphism can exist when a subclass modifies or customizes the behavior of a method inherited from its superclass in order to meet the special requirements of objects instantiated from the subclass. This is known as *overriding* a method and requires the use of the **override** keyword in ActionScript.

Override methods differently for different subclasses

ActionScript supports the notion of *overriding* a method inherited from a superclass to cause the named method to behave differently when called on objects of different subclasses, each of which extends the same superclass and overrides the same method name.

Example - compute the area of different geometrical shapes

For example, consider the computation of the area of a geometrical shape in a situation where the type of geometrical shape is not known when the program is compiled. Polymorphism is a tool that can be used to handle this situation.

Circle and Rectangle extend Shape

Assume that classes named **Circle** and **Rectangle** each extend a class named **Shape** . Assume that the **Shape** class defines a method named **area** . Assume further that the **area** method is properly overridden in the **Circle** and **Rectangle** classes to return the correct area for a circle or a rectangle respectively.

Three types of objects

In this case, a **Circle** object is a **Shape** object because the **Circle** class extends the **Shape** class. Similarly, a **Rectangle** object is also a **Shape** object.

Therefore, an object of the **Shape** class, the **Circle** class, or the **Rectangle** class can be instantiated and any one of the three can be saved in a variable of type **Shape** .

Flip a virtual random coin

Assume that the program flips a virtual coin and, depending on the outcome of the flip, instantiates an object of either the **Circle** class or the **Rectangle** class and saves it in a variable of type **Shape** . Assuming that the coin flip is truly random, the compiler cannot possibly know at compile time which type of object will be stored in the variable at runtime.

Two versions of the area method

Regardless of which type of object is stored in the variable, the object will contain two versions of the method named **area** . One version is the version that is defined in the **Shape** class, and this version will be the same regardless of whether the object is a circle or a rectangle.

Also, this version can't return a valid area because a general shape doesn't have a valid area. However, if the area method is defined to return a value, even this version must return a value even if it isn't valid. *(Other programming languages get around this problem with something called an abstract class, which isn't allowed in ActionScript 3.)*

The other version of the **area** method will be different for a **Circle** object and a **Rectangle** object due simply to the fact the algorithm for computing the area of a circle is different from the algorithm for computing the area of a rectangle.

Call the area method on the object

If the program calls the **area** method on the object stored in the variable of type **Shape** at runtime, the correct version of the **area** method will be selected and executed and will return the correct area for the type of object stored in that variable. This is *runtime polymorphism* based on method overriding.

A more general description of runtime polymorphism

A reference variable of a superclass type can be used to reference an object instantiated from any subclass of the superclass.

If an overridden method in a subclass object is called using a superclass-type reference variable, the system will determine, at runtime, which version of the method to use based on the true type of the object, and not based on the type of reference variable used to call the method.

The general rule

The type of the reference determines the names of the methods that can be called on the object. The actual type of the object determines which of possibly several methods having the same name will be executed.

Selection at runtime

Therefore, it is possible *(at runtime)* to select among a family of overridden methods and determine which method to execute based on the type of the subclass object pointed to by the superclass-type reference when the overridden method is called on the superclass-type reference.

Runtime polymorphism

In some situations, it is possible to identify and call an overridden method at runtime that cannot be identified at compile time. In those situations, the identification of the required method cannot be made until the program is actually running. This is often referred to as *late binding* , *dynamic binding* , or *run-time polymorphism* .

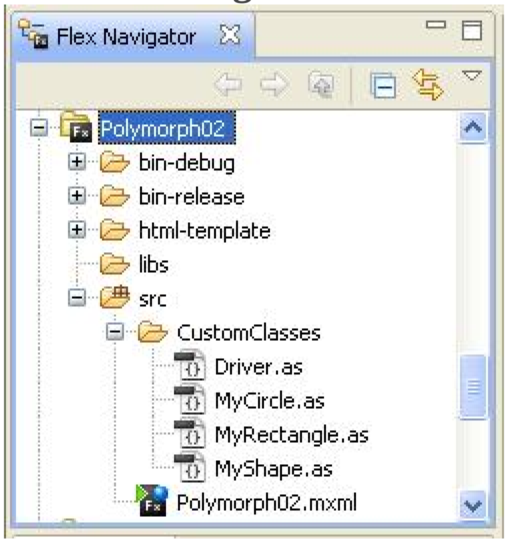
Preview

Project file structure

Figure 1 shows the project file structure for the Flex project named **Polymorph02** .

File structure for Polymorph02.

Figure



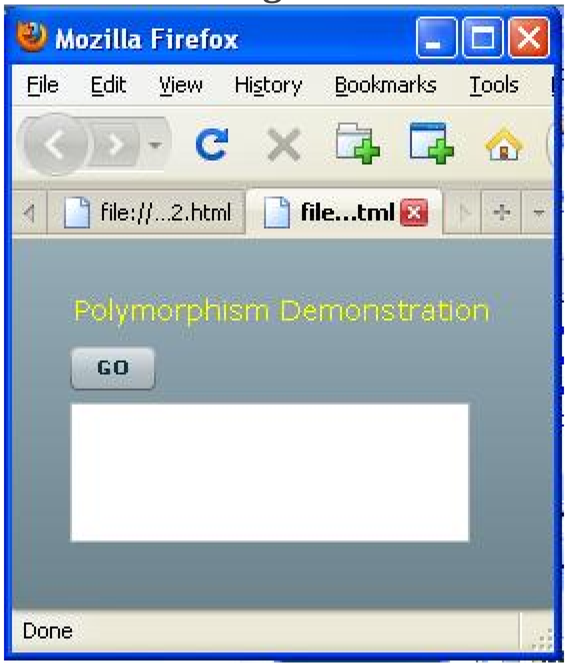
```
Flex Navigator
Polymorph02
  bin-debug
  bin-release
  html-template
  libs
  src
    CustomClasses
      Driver.as
      MyCircle.as
      MyRectangle.as
      MyShape.as
    Polymorph02.mxml
```

File structure for
Polymorph02.

The image in Figure 1 was captured from the Flex Navigator panel of Flex Builder 3.

Program output at startup

Figure 2 shows the program output at startup.
Program output at startup.
Figure


Mozilla Firefox
File Edit View History Bookmarks Tools
file://...2.html
file...tml
Polymorphism Demonstration
GO
Done

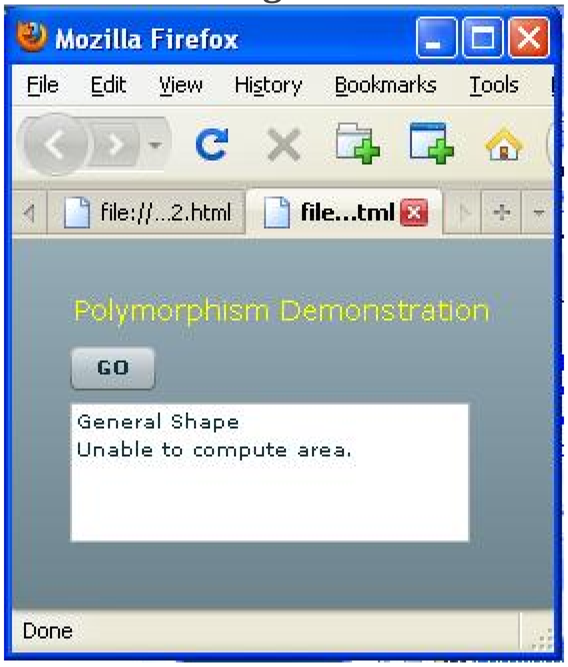
Program output at startup.

The program GUI

As you can see, the program GUI consists of a label, a button and an empty text area. Each time the user clicks the button, an object is instantiated, and

information about that object *(similar to the information shown in Figure 3)* is displayed in the text area.
Sample output from area method of a MyShape object.

Figure


Mozilla Firefox
File Edit View History Bookmarks Tools
file://...2.html
file...tml
Polymorphism Demonstration
GO
Done

```
General Shape
Unable to compute area.
```

Sample output from area method of a MyShape object.

Program class definitions

This program defines the following four classes as shown in the folder named **CustomClasses** in Figure 1:

- **Driver**
- **MyCircle**
- **MyRectangle**
- **MyShape**

As the name implies, the **Driver** class is the driver for the entire program. For example, it creates the GUI shown in Figure 2 at startup and updates the GUI each time the user clicks the **GO** button as shown in Figure 3.

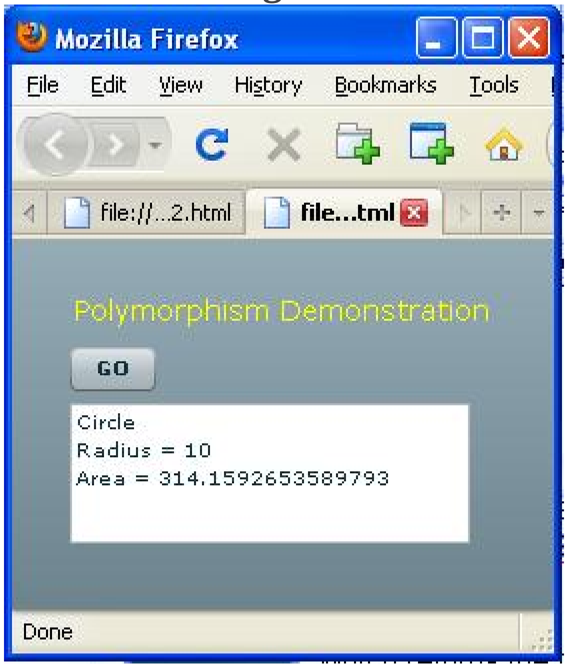
The classes named **MyCircle** and **MyRectangle** each extend the class named **MyShape** . Therefore, specialized shape objects can be instantiated from these two classes.

The class named **MyShape** defines an instance method named **area** , which returns the text shown in Figure 3 each time it is executed.

Sample output from area method of a MyCircle object

The class named **MyCircle** overrides the inherited **area** method to return a text string similar to that shown in Figure 4 each time it is executed.
Sample output from area method of a MyCircle object.

Figure


Mozilla Firefox
File Edit View History Bookmarks Tools
file://...2.html
file...tml
Polymorphism Demonstration
GO
Done

```
Circle
Radius = 10
Area = 314.1592653589793
```

Sample output from area method of a MyCircle object.

The radius is a random value

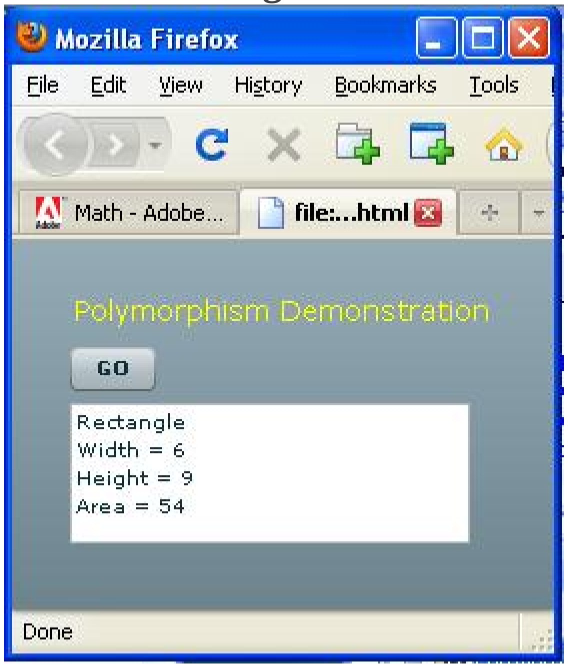
Note, however, that the value of radius is established from a random number generator each time an object of the **MyCircle** class is instantiated, so the actual values for *Radius* and *Area* in Figure 4 will change each time the button is clicked.

Sample output from area method of a MyRectangle object

The class named **MyRectangle** also overrides the inherited **area** method to return a text string similar to that shown in Figure 5 each time it is executed.

Sample output from area method of a MyRectangle object.

Figure



```
Rectangle
Width = 6
Height = 9
Area = 54
```

Sample output from area method of a MyRectangle object.

Width and height are random values

Once again, however, the values of Width and Height are established from a random number generator each time an object of the **MyRectangle** class is instantiated, so the actual values for *Width* , *Height* , and *Area* in Figure 5 will change each time the button is clicked.

A click event handler

A **click** event handler is registered on the button shown in Figure 5. Each time the button is clicked, the event handler uses the output from a random number generator to instantiate an object of one of the **following classes** and saves it in a variable of type **MyShape** .

- **MyShape**
- **MyCircle**
- **MyRectangle**

Multiple versions of the method named area

As I explained earlier , an object of the **MyShape** class will contain one version of the **area** method, but objects of the other two classes will each contain two versions of the **area** method.

One version of the method is common among all three objects, and that is the version that returns the text shown in Figure 3.

However, the other version in each of the **MyCircle** and **MyRectangle** classes is unique to its class returning values similar to those shown in Figure 4 and Figure 5.

No selection requirement for object of type MyShape

When the **area** method is called on an object of the **MyShape** class, there is no requirement to select a *"correct"* version of the method because that object only contains one version of the method.

Polymorphism kicks in...

Because the **area** method is defined in the **MyShape** class and overridden in the **MyCircle** class and the **MyRectangle** class, and because the objects of the **MyCircle** class and the **MyRectangle** class are saved as type **MyShape** , polymorphism kicks in when the **area** method is called on those objects. The overridden method is selected for execution in those cases where the object contains an overridden version of the method.

Could easily expand the shape system

Because of polymorphism, I could easily expand this system of shapes by defining new classes for new shapes *(such as triangle, octagon, hexagon, etc.)* , without any requirement to modify the **MyShape** class, the **MyCircle** class, or the **MyRectangle** class.

Assuming that I cause the classes for the new shapes to extend the **MyShape** class and properly override the **area** method to return the correct values, I could then instantiate objects of the new classes, save them as type **MyShape** , call the **area** method on those new objects and expect the correct version of the **area** method to be executed.

The most powerful concept...

This is runtime polymorphism in all its glory, and is probably the most powerful concept in all of OOP.

Discussion and sample code

Will discuss in fragments

I will discuss the code in the five files shown in Figure 1 in fragments. Complete listings of those files are provided in Listing 9 through Listing 13 near the end of the lesson.

The file named Polymorph02.mxml

The complete MXML code for the file named **Polymorph02.mxml** is provided in Listing 1 and also in Listing 9.

Example:
Source code for the file named Polymorph02.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<!--Illustrates polymorphism.-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

In keeping with my objective of emphasizing ActionScript programming rather than Flex MXML programming in this series of tutorial lessons, this program is written almost entirely in ActionScript. The MXML file serves simply as a launch pad for the program by instantiating an object of the class named **Driver** .

The file named Driver.as

A complete listing of this file is provided in Listing 10 near the end of the lesson. The class named **Driver** begins in Listing 2.

Example:
Beginning of the class named Driver.

```
package CustomClasses{
  import flash.events.*;

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.controls.TextArea;

  public class Driver extends VBox{
    private var textArea:TextArea = new
TextArea();
```

A Driver object is a VBox object

The class named **Driver** extends the **VBox** class. As a result, components added to the objects are arranged vertically with left justification as shown in Figure 2.

The TextArea object

Listing 2 instantiates the **TextArea** object shown in Figure 2 and saves its reference in an instance variable named **textArea** . The reference was saved in an instance variable to make it available to both the constructor and a **click** event handler method later.

The constructor for the class named Driver

The constructor for the class named **Driver** is shown in Listing 3.

Example:
The constructor for the class named Driver.

```
    public function Driver(){//constructor
      var label:Label = new Label();
      label.text = "Polymorphism Demonstration";
      label.setStyle("fontSize",14);
      label.setStyle("color",0xFFFF00);
      addChild(label);

      var button:Button = new Button();
      button.label = "GO";
      addChild(button);

      textArea.width = 200;
      textArea.height = 70;
      addChild(textArea);

      button.addEventListener(

MouseEvent.CLICK,buttonHandler);
    }//end constructor
```

Nothing new here

I don't believe there is anything in Listing 3 that you haven't seen in previous lessons. The code in the constructor simply constructs the GUI pictured in Figure 2.

A click event handler method

Probably the most important thing to note about listing 3 is the registration of the **click** event handler method named **buttonHandler** on the button shown in Figure 2. Once the GUI is constructed, further action occurs only when the user clicks the button, causing the method named **buttonHandler** to be executed.

Beginning of the method named buttonHandler

The method named **buttonHandler** begins in Listing 4. This method is executed each time the user clicks the button shown in Figure 2.

Example:
Beginning of the method named buttonHandler.

```
    private function buttonHandler(

event:MouseEvent):void{
      var randomChoice:Number = Math.random();

      var radius:uint = uint(10*Math.random() +
1);
      var width:uint = uint(10*Math.random() + 1);
      var height:uint = uint(10*Math.random() +
1);
```

Four local variables

The most interesting thing in Listing 4 is the declaration and population of the following four local variables:

- **randomChoice**
- **radius**
- **width**
- **height**

A random value

The variable named **randomChoice** is of type **Number** and is populated with the return value from a call to the **random** method of the **Math** class. The documentation tells us that this method:

> *"Returns a pseudo-random number n, where 0 LTE n LT 1. The number returned is calculated in an undisclosed manner, and*

> *pseudo-random because the calculation inevitably contains some element of non-randomness."*

Note that LTE and LT represent *"less than or equal"* and *"less than"* respectively. *(Lots of problems arise from including angle brackets and ampersands in html text so I avoid doing that when I can.)*

A fractional random value

Thus, the return value is a fractional value that is greater than or equal to zero and less than 1.0. This value will be used later to decide which type of object to instantiate.

Three more random values

The last three variables in the above list are also populated with random values, but not with fractional values in the range from 0 to 0.999... Instead, these variables are populated with unsigned integer values in the range of 1 to 10 inclusive. This is accomplished by multiplying the original random value by 10, adding 1 to that product, and casting the result to an unsigned integer.

A local variable of type MyShape

Listing 5 begins by declaring a variable of type **MyShape** that will be used to hold a reference to an object instantiated from one of the following classes: **MyShape** , **MyCircle** , or **MyRectangle** .

Example:
Instantiate an object.

```
      var myShape:MyShape;

      if(randomChoice < 0.33333){
        myShape = new MyShape();
      }else if(randomChoice < 0.66666){
        myShape = new MyCircle(radius);
      }else{
        myShape = new MyRectangle(width,height);
      }//end else
```

Instantiate an object

Then Listing 5 uses the random value stored in the variable named **randomChoice** to make a decision and to instantiate an object of one of the three classes listed above . The decision as to which type of object to instantiate and store in the variable named **myShape** is completely random and completely unknown to the compiler when the program is compiled.

Random values for radius, width, and height

Note also that Listing 5 also uses the other values stored in the other variables in the above list to specify the radius of the circle, or to specify the width and the height of the rectangle.

Call the area method on the object

Finally, *(and this is the essence of runtime polymorphism)* , Listing 6 calls the **area** method on the randomly instantiated object and writes the return value into the text area shown in Figure 2.

Example:
Call the area method on the object.

```
      textArea.text = myShape.area();
    }//end buttonHandler

  }//end class
}//end package
```

And the result will be...

The result will be similar to Figure 3, Figure 4, or Figure 5, depending on which type of object is instantiated and depending on the random values passed for radius, width, and height.

The end of the class

Listing 6 also signals the end of the class named **Driver** .

The file named MyShape.as

The class named **MyShape** is shown in its entirety in Listing 7, and also in Listing 11 near the end of the lesson.

Example:
The class named MyShape.

```
package CustomClasses{
  public class MyShape{

    public function area():String{
      return "General Shape\n" +
             "Unable to compute area.";
    }// end area method

  }//end class
}//end package
```

The sole purpose

The sole purpose of the class named **MyShape** is to serve as the root of a hierarchy of more specialized shape classes and to provide a default version of the **area** method that can be overridden in the subclasses.

The class accomplishes those things well, and beyond that, there isn't much to say about the code in Listing 7. Note that the class doesn't even define a constructor but instead uses the default constructor that is provided by the compiler.

The file named MyCircle.as

The class named **MyCircle** is shown in its entirety in Listing 8, and also in Listing 12 near the end of the lesson.

Example:
The class named MyCircle.

```
package CustomClasses{
  public class MyCircle extends MyShape{
    private var radius:Number;

    public function MyCircle(radius:Number)
{//constructor
      this.radius = radius;
    }//end constructor

    override public function area():String{
      return "Circle\n" +
             "Radius = " + radius + "\n" +
             "Area = " + Math.PI * radius *
radius;
    }//end area
  }//end class
}//end package
```

The class named **MyCircle** is only slightly more complicated than the class named **MyShape** .

The constructor

The constructor for the class named **MyCircle** receives an incoming value for the radius and saves that value in a private instance variable named **radius** . Saving the value in an instance variable makes it available to the **area** method that will be executed later.

Type coercion

Recall from Listing 4 and Listing 5 that the value that is actually passed to the constructor is of type **uint** , which is an unsigned integer. That value is coerced to type **Number** by passing it as an argument that is defined to be of type **Number** .

An overridden area method

Note that the method named **area** is declared to be an **override.** This syntax is required by ActionScript when a method in a subclass overrides an inherited method.

Return a String object

The code in the **area** method concatenates several individual strings *(including the computed area of the circle)* into a single **String** object and returns a reference to that object. A sample of the returned value is shown displayed in the text area of Figure 4.

Concatenation of strings with numeric values

The computed value of the area and the stored value of the radius are both numeric values. When a numeric value is concatenated with a string, the numeric value is coerced into a string of characters, decimal points, etc., and the two strings are concatenated into a single string.

Computation of the area

As you can see in Listing 8, the area of the circle is computed as the square of the radius multiplied by the constant PI. *(Hopefully you recall that formula from your high school geometry class.)* As mentioned earlier, the resulting area value is concatenated with the string to its left where it becomes part of the larger **String** object that is returned by the method.

The file named MyRectangle.as

The class named **MyRectangle** is so similar to the class named **MyCircle** that it doesn't warrant a detailed explanation. A complete listing of the file is provided in Listing 13 near the end of the lesson.

The constructor receives and saves numeric values for the width and the height of the rectangle.

The overridden **area** method computes the area as the product of the width and the height and concatenates that information into a returned **String** object where it is displayed in the format shown in Figure 5.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 9 through Listing 13. Use that code to create a Flex project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listing

Complete listings of the Flex and ActionScript files used in this program are provided in Listing 9 through Listing 13 below.

Example:
Listing of the file named Polymorph02.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<!--Illustrates polymorphism.-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Example:
Listing of the file named Driver.as.

```
package CustomClasses{
  import flash.events.*;

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.controls.TextArea;

  public class Driver extends VBox{
    private var textArea:TextArea = new
TextArea();


    public function Driver(){//constructor
      var label:Label = new Label();
      label.text = "Polymorphism Demonstration";
      label.setStyle("fontSize",14);
      label.setStyle("color",0xFFFF00);
      addChild(label);

      var button:Button = new Button();
      button.label = "GO";
      addChild(button);

      textArea.width = 200;
      textArea.height = 70;
      addChild(textArea);

      button.addEventListener(

MouseEvent.CLICK,buttonHandler);
    }//end constructor

    private function buttonHandler(

event:MouseEvent):void{
      var randomChoice:Number = Math.random();
      var radius:uint = uint(10*Math.random() +
1);
      var width:uint = uint(10*Math.random() + 1);
      var height:uint = uint(10*Math.random() +
1);
      var myShape:MyShape;
      if(randomChoice < 0.33333){
        myShape = new MyShape();
      }else if(randomChoice < 0.66666){
        myShape = new MyCircle(radius);
      }else{
        myShape = new MyRectangle(width,height);
      }
      textArea.text = myShape.area();
    }//end buttonHandler

  }//end class
}//end package
```

Example:
Listing of the file named MyShape.as.

```
package CustomClasses{
  public class MyShape{

    public function area():String{
      return "General Shape\n" +
             "Unable to compute area.";
    }// end area method
  }//end class

}//end package
```

Example:
Listing of the file named MyCircle.as.

```
package CustomClasses{
  public class MyCircle extends MyShape{
    private var radius:Number;

    public function MyCircle(radius:Number)
{//constructor
      this.radius = radius;
    }//end constructor

    override public function area():String{
      return "Circle\n" +
             "Radius = " + radius + "\n" +
             "Area = " + Math.PI * radius *
radius;
    }//end area
  }//end class
}//end package
```

Example:
Listing of the file named MyRectangle.as

```
package CustomClasses{
  public class MyRectangle extends MyShape{
    private var width:Number;
    private var height:Number;

    public function MyRectangle(
                width:Number,height:Number)
{//constructor
      this.width = width;
      this.height = height;
    }//end constructor

    override public function area():String{
      return "Rectangle\n" +
             "Width = " + width + "\n" +
             "Height = " + height + "\n" +
             "Area = " + width * height;
    }//end area
  }//end class
}//end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Polymorphism - The Big Picture
- Files:
  - ActionScript0110\ActionScript0110.htm
  - ActionScript0110\Connexions\ActionScriptXhtml0110.htm

-end-

Interface Polymorphism - The Big Picture
Learn how to use runtime polymorphism based on the ActionScript interface.

Note: Click Interface01 to run this ActionScript program. (Click the "Back" button in your browser to return to this page.)

Table of Contents

- Miscellaneous

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a continuing series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

The three main characteristics of an object-oriented program

Object-oriented programs exhibit three main characteristics:

- Encapsulation
- Inheritance
- Polymorphism

There are two different ways to implement polymorphism:

- Polymorphism based on class inheritance
- Polymorphism based on interface inheritance

I explained encapsulation, inheritance, and polymorphism based on class inheritance in previous lessons. *(See Baldwin's ActionScript programming website .)*

I will explain and illustrate polymorphism based on interface inheritance in this lesson.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

What is an ActionScript interface?

In an earlier lesson titled *"Inheritance - The Big Picture"* , I told you that unlike C++,

> *"ActionScript 3 does not support multiple inheritance. Instead it supports a different mechanism called an interface that provides most of the benefits of multiple inheritance without most of the problems."*

I promised to explain the ActionScript interface in a future lesson, and that time has come.

What does the documentation have to say?

According to About interfaces ,

> *"Interfaces are a type of class that you design to act as an outline for your components. When you write an interface, you provide only the names of public methods rather than any implementation. For example, if you define two methods in an interface and then implement that interface, the implementing class must provide implementations of those two methods.*
>
> *Interfaces in ActionScript can declare methods and properties only by using setter and getter methods; they cannot specify constants. The benefit of interfaces is that you can define a contract that all classes that implement that interface must follow. Also, if your class implements an interface, instances of that class can also be cast to that interface."*

According to Interfaces ,

> *"An interface is a collection of method declarations that allows unrelated objects to communicate with one another...*

> *Interfaces are based on the distinction between a method's interface and its implementation. A method's interface includes all the information necessary to invoke that method, including the name of the method, all of its parameters, and its return type. A method's implementation includes not only the interface information, but also the executable statements that carry out the method's behavior. An interface definition contains only method interfaces, and any class that implements the interface is responsible for defining the method implementations...*
>
> *Another way to describe an interface is to say that it defines a data type just as a class does. Accordingly, an interface can be used as a type annotation, just as a class can. As a data type, an interface can also be used with operators, such as the is and as operators, that require a data type. Unlike a class, however, an interface cannot be instantiated. This distinction has led many programmers to think of interfaces as abstract data types and classes as concrete data types."*

According to this author...

An interface is like a class in which all of the methods are abstract, meaning that only their user interface is declared. Their behavior or implementation is not defined. *(Recall from an earlier lesson that it is not possible to declare an abstract method in an ActionScript class. All methods must be defined as concrete methods in ActionScript classes.)*

A form of multiple inheritance

In ActionScript, a class can inherit from *(extend)* only one other class. However, a class can inherit from *(implement)* any number of interfaces. Furthermore, each interface can extend any number of other interfaces.

Therefore, an ActionScript class can inherit any number of *concrete* methods from one superclass and can inherit any number of *abstract* methods from any number of interfaces.

Concrete methods are required

Any class that inherits an abstract method must provide a concrete definition for the method or the class cannot be compiled.

ActionScript classes cannot be declared abstract. Therefore, the abstract method cannot be passed down the class hierarchy for definition by a subclass as is the case in Java. The concrete version must be defined in the class in which it is inherited. This means that it must be defined in the class that implements the interface.

What is a concrete method?

As a minimum, a concrete method is a method signature followed by a pair of matching curly braces *(possibly containing a return statement)* . Normally the body of the method is coded between the curly braces defining the behavior of the method.

If the return type is void, the matching curly braces may be empty. In that case, the concrete method exhibits no observable behavior. It returns immediately without doing anything when it is called.

If the return type is not void, there must be a return statement that returns a value of the correct type.

Do empty methods have any value?

It is not uncommon for a class to implement an interface for which some of the interface methods are of no interest with regard to an object of the new class. However, the implementing class must provide concrete definitions for all of the methods inherited from the interface.

In that case, it is common practice to simply define the uninteresting methods as empty methods in the new class. If they ever do get called, they will simply return immediately without doing anything.

Preview

The program named **Interface01** that I will explain in this lesson is an update of the program named **Polymorph02** that I explained in the earlier lesson titled *"Polymorphism - The Big Picture" (see* *Baldwin's ActionScript programming website* *).*

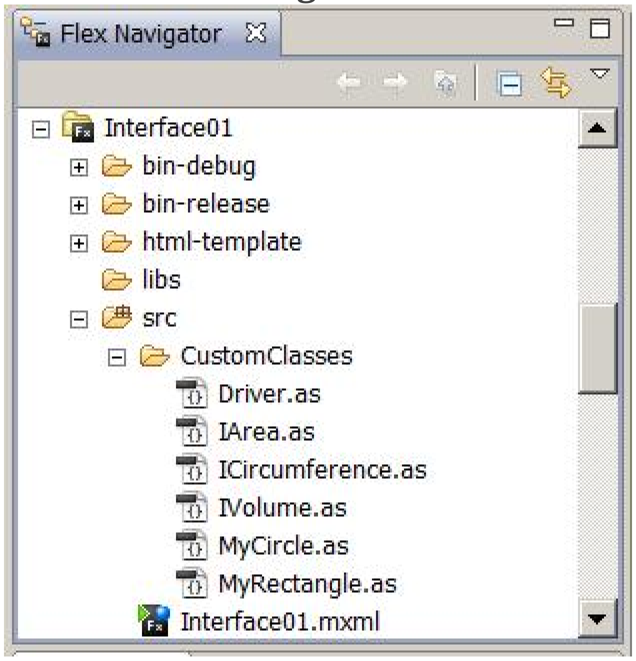
The earlier program illustrated runtime polymorphism based on class inheritance and the overriding of inherited concrete methods.

This program will illustrate runtime polymorphism based on interface inheritance and the concrete definition of inherited abstract methods.

The project file structure

The project file structure is shown in Figure 1.
Project file structure.

Figure



Project file structure.

Six ActionScript files

This project consists of one MXML file named **Interface01** , three class files, and three interface files. The class files are named:

- **MyRectangle.as**
- **MyCircle.as**
- **Driver.as**

The interface files are named:

- **IVolume.as**
- **ICircumference.as**
- **IArea.as**

Interface can declare any number of methods

An interface can declare any number of methods including setter and getter methods. As you will see later, the interface named **IArea** declares two methods named **area** and **id** .

The interface named **IVolume** declares one method named **volume** . Likewise, the interface named **ICircumference** declares one method named **circumference** .

The rules of the road

Any number of classes can implement the same interface. This makes it possible to treat objects instantiated from different classes as the common interface type.

A class can implement any number of interfaces. In this program, however, the classes named **MyRectangle** and **MyCircle** each implement only one interface named **IArea** .

Can extend any number of interfaces

An interface can extend any number of other interfaces. In this program, the **IArea** interface extends both the **IVolume** interface and the **ICircumference** interface.

The bottom line

Implementing the **IArea** interface causes both the **MyRectangle** class and the **MyCircle** class to inherit the following abstract methods:

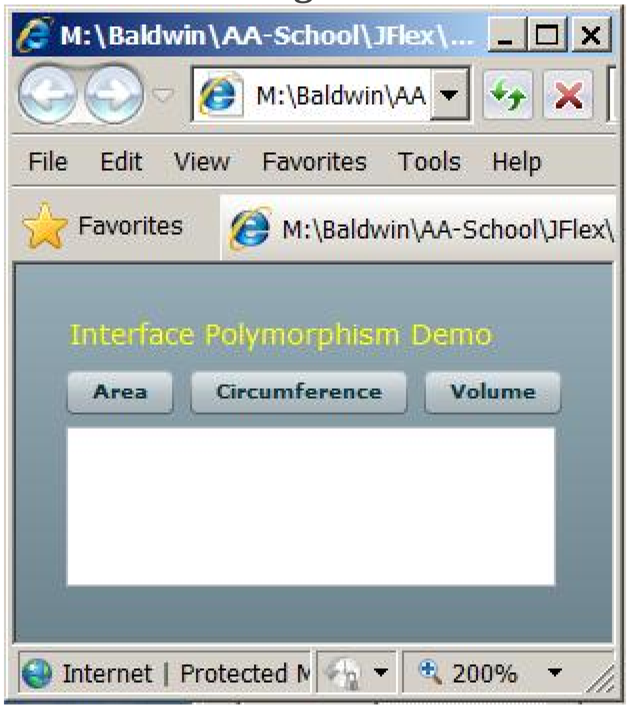
- **area**
- **id**
- **circumference**
- **volume**

The first two methods are inherited directly from the interface named **IArea** . The last two are inherited from the interfaces named **ICircumference** and **IVolume** by way of **IArea** .

Each of the two classes must provide concrete definitions for all four of the inherited abstract methods.

Program output at startup

Figure 2 shows the program output at startup.
Program output at startup.
Figure


M:\Baldwin\AA-School\JFlex\...
M:\Baldwin\AA
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JFlex\
Interface Polymorphism Demo
Area
Circumference
Volume
Internet | Protected M
200%

Program output at startup.

The program GUI

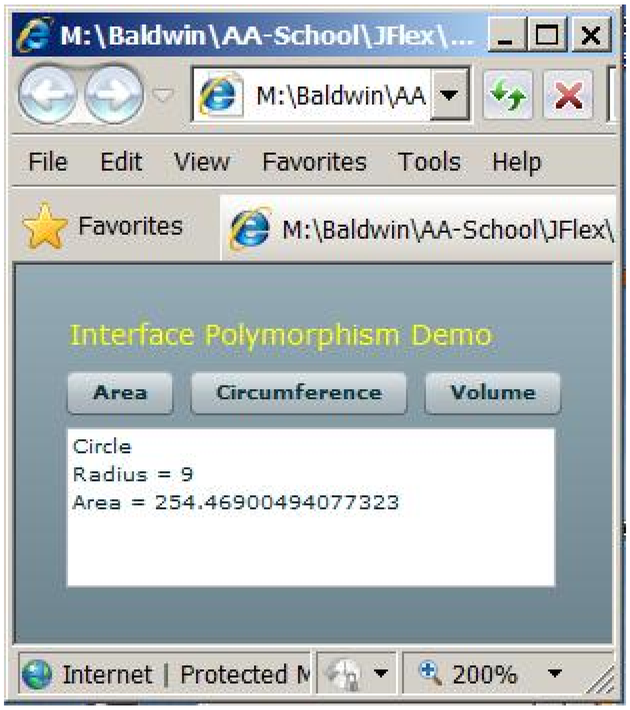
The program GUI consists of one *label* , three *buttons* and a *text area* . When a button is clicked, a random process is used to instantiate an object of either the **MyCircle** class or the **MyRectangle** class. The new object's reference is saved in a variable of the interface type **IArea** . This is possible because both classes implement the common **IArea** interface.

Call methods using the reference of type IArea

Then, as a result of a click on a button, the program calls the **id** method on the object whose reference is stored in the variable of type **IArea** . Depending on which button is clicked, the program then calls either the **area** method, the **circumference** method, or the **volume** method on the object.

Output after clicking the Area Button for circle

Clicking the button labeled Area, for example causes the output to change to something similar to either Figure 3 or Figure 4. An output similar to Figure 3 is produced if the random process instantiates an object of the **MyCircle** class.
Output after clicking the Area Button for circle.
Figure

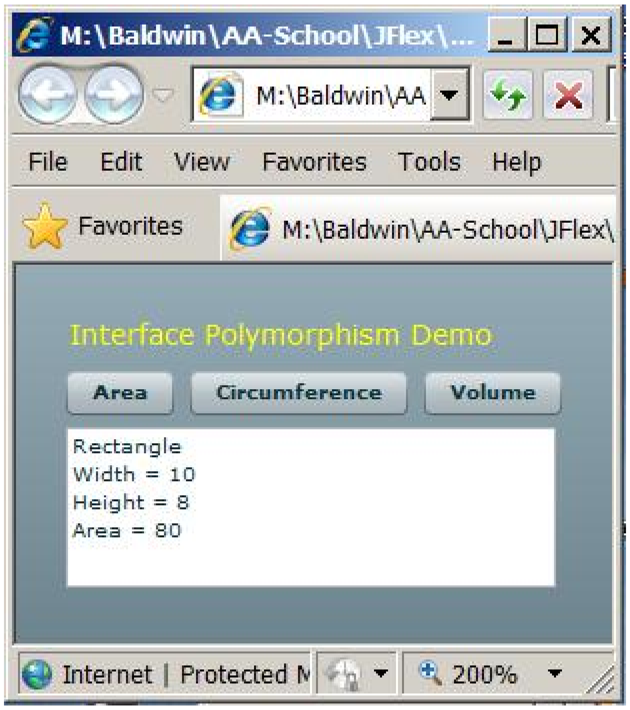

M:\Baldwin\AA-School\JFlex\...
M:\Baldwin\AA
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JFlex\
Interface Polymorphism Demo
Area
Circumference
Volume
Internet | Protected M
200%

```
Circle
Radius = 9
Area = 254.46900494077323
```

Output after clicking the Area Button for circle.

Output after clicking the Area Button for rectangle

An output similar to Figure 4 is produced if the random process instantiates an object of the **MyRectangle** class.
Output after clicking the Area Button for rectangle.
Figure


M:\Baldwin\AA-School\JFlex\...
M:\Baldwin\AA
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JFlex\
Interface Polymorphism Demo
Area
Circumference
Volume
Internet | Protected M
200%

```
Rectangle
Width = 10
Height = 8
Area = 80
```

Output after clicking the Area Button for rectangle.

Random radius, width, and height

I use the word similar because a random process is also used to establish the radius for the circle and to establish the width and height for the rectangle.

Both the **MYCircle** object and the **MyRectangle** object contain concrete versions of the four methods listed above . However, the behavior of those four methods in one object is different from the behavior of the four methods having the same names in the other object.

Polymorphism kicks in

The compiler can't possibly know which type of object will be instantiated as a result of the random process following each button click when the program is compiled. Therefore, the decision as to which set of methods to call as a result of each button click cannot be determined until runtime. This is the essence of *runtime polymorphism* .

The cardinal rule

The type of the object's reference determines which set of method names can be called on that reference. In this case, the set consists of the four methods declared in and inherited into the interface named **IArea** , which are shown in the above list .

The type of the object determines which method from the set of allowable names is actually executed.

Run the program

You can run the program to see the outputs produced by repeatedly clicking each of the three buttons in Figure 2.

Discussion and sample code

Will discuss in fragments

I will break the longer files in this application down and discuss them in fragments. Complete listings of all of the files are provided in Listing 10 through Listing 16 near the end of the lesson.

The file named Interface01.mxml

In keeping with my plan to emphasize ActionScript over Flex in this series of lessons, the MXML file for this application is very simple, instantiating only a single object of type **Driver** . A listing of the MXML file is provided in Listing 10 near the end of the lesson.

The file named Driver.as

The class named **Driver** begins in Listing 1. A complete listing of the file is provided in Listing 11 near the end of the lesson.

Example:
Beginning of the class named Driver.

```
package CustomClasses{
  import flash.events.*;

  import mx.containers.HBox;
  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.controls.TextArea;

  public class Driver extends VBox{
    private var textArea:TextArea = new
TextArea();
    private var myShape:IArea;

    private var randomChoice:Number;
    private var radius:uint;
    private var rectWidth:uint;
    private var rectHeight:uint;
```

Listing 1 declares several new instance variables. The most interesting variable is named **myShape** because it is declared to be of type **IArea** , which is the name of an interface.

References to objects of the classes **MyCircle** and **MyRectangle** will be stored in this variable.

Beginning of the constructor for the Driver class

The constructor for the **Driver** class begins in Listing 2.

Example:
Beginning of the constructor for the Driver class.

```
    public function Driver(){//constructor
      var label:Label = new Label();
      label.text = "Interface Polymorphism Demo";
      label.setStyle("fontSize",14);
      label.setStyle("color",0xFFFF00);
      addChild(label);

      //Put three buttons in an HBox
      var hbox:HBox = new HBox();
      addChild(hbox);

      var areaButton:Button = new Button();
      areaButton.label = "Area";
      hbox.addChild(areaButton);

      var circButton:Button = new Button();
      circButton.label = "Circumference";
      hbox.addChild(circButton);

      var volumeButton:Button = new Button();
      volumeButton.label = "Volume";
      hbox.addChild(volumeButton);

      //Put the text area below the HBox.
      textArea.width = 245;
      textArea.height = 80;
      addChild(textArea);
```

You shouldn't find any surprises in Listing 2. The code in Listing 2 simply constructs the GUI shown in Figure 2, placing the label, the buttons, and the text area in their respective locations.

The remainder of the constructor for the Driver class

The remainder of the constructor for the **Driver** class is shown in Listing 3.

Example:
The remainder of the constructor for the Driver class.

```
      //Register a click event handler on each of
the
      // buttons
      areaButton.addEventListener(

MouseEvent.CLICK,areaButtonHandler);

      circButton.addEventListener(

MouseEvent.CLICK,circButtonHandler);

      volumeButton.addEventListener(

MouseEvent.CLICK,volumeButtonHandler);
    }//end constructor
```

Listing 3 registers a **click** event handler on each of the buttons shown in Figure 2.

The getRandomValues method

The method named **getRandomValues** is shown in Listing 4.

Example:
The getRandomValues method.

```
    //Local utility method for getting and saving
four
    // random values.
    private function getRandomValues():void{
      randomChoice = Math.random();
      radius = uint(10*Math.random() + 1);
      rectWidth = uint(10*Math.random() + 1);
      rectHeight = uint(10*Math.random() + 1);
    }//end getRandomValues
```

This method is called by each of the event handler methods to get and save four random values that are subsequently used by the event handler. The random values are saved in the instance variables that are declared in Listing 1.

The event handler method named areaButtonHandler

The **click** event handler that is registered on the button labeled **Area** in Figure 2 is shown in Listing 5.

Example:
The event handler method named areaButtonHandler.

```
    //Define click event handler methods.
    private function areaButtonHandler(

event:MouseEvent):void{
      getRandomValues();

      if(randomChoice < 0.5){
        myShape = new MyCircle(radius);
      }else{
        myShape = new
MyRectangle(rectWidth,rectHeight);
      }//end else

      textArea.text = myShape.id() +
myShape.area();
    }//end areaButtonHandler
```

Decide between two classes

The code in Listing 5 uses the random value stored in **randomChoice** to decide whether to instantiate an object of the class named **MyCircle** or the class named **MyRectangle** .

Store object's reference as type IArea

The object's reference is stored in the instance variable named **myShape** , which is type **IArea** . Storage of the reference in a variable of that type is possible only because both objects implement the interface named **IArea** .

Call the id method using the reference

Then the code in Listing 5 uses the reference stored in **myShape** to call the **id** method and the **area** method on the object. The returned values are used to construct a new text string for the text area shown in Figure 3.

How is this possible?

Calling these methods using the reference of type **IArea** is possible only because abstract versions of both methods are inherited into both classes from the interface named **IArea** .

The other two click event handler methods

The **click** event handler methods that are registered on the **Circumference** button and the **Volume** button in Figure 2 are shown in Listing 6.

Example:
The other two click event handler methods.

```
    private function circButtonHandler(

event:MouseEvent):void{
      getRandomValues();

      if(randomChoice < 0.5){
        myShape = new MyCircle(radius);
      }else{
        myShape = new
MyRectangle(rectWidth,rectHeight);
      }//end else
      textArea.text = myShape.id() +

myShape.circumference();
    }//end circButtonHandler


    private function volumeButtonHandler(

event:MouseEvent):void{
      getRandomValues();

      if(randomChoice < 0.5){
        myShape = new MyCircle(radius);
      }else{
        myShape = new
MyRectangle(rectWidth,rectHeight);
      }//end else
      textArea.text = myShape.id() +
myShape.volume();
    }//end circButtonHandler

  }//end class
}//end package
```

The same methodology

The methodology behind these two event handlers is the same as the methodology behind the event handler method shown in Listing 5.

The main differences between the event handlers

The main differences appear in the two statements that begin with **"textArea.text = "** . In addition to using the reference of type **IArea** to call the **id** method on the objects, these two event handler methods call the **circumference** method or the **volume** method on the objects. Once again, this is possible only because the objects inherit those methods from the **IArea** interface.

The end of the Driver class

Listing 6 also signals the end of the class named **Driver** .

The file named IArea.as

Since I mentioned the interface named **IArea** several times above, I will discuss it next.

The interface named **IArea** is shown in Listing 7.

Example:

The interface named IArea.

```
package CustomClasses{
  public interface IArea extends
IVolume,ICircumference{
    function area():String;
    function id():String;
  }//end interface
}//end package
```

General information about an interface

The syntax for an interface looks a lot like the syntax for a class. However, the keyword **class** is replaced by the keyword **interface** .

An interface may contain only method declarations *(with no bodies)* and setter and getter method declarations. All method declarations are implicitly **public** , and the concrete definition of an interface method in a class must be declared **public** .

You cannot instantiate an object of an interface.

An interface cannot extend a class, but can extend any number of other interfaces.

IArea declares two methods and extends two interfaces

The interface shown in Listing 7 declares the abstract methods named **area** and **id** . It also extends the interfaces named **IVolume** and **ICircumference** .

Concrete method definitions are required

Any class that implements the interface named **IArea** must provide concrete definitions for the methods named **id** and **area** .

The class must also provide concrete definitions for any methods inherited into **IArea** from the interfaces that it extends. The class also inherits those methods by way of **IArea** .

The file named IVolume.as

The interface named **IVolume** is shown in its entirety in Listing 8.

Example:
The interface named IVolume.

```
package CustomClasses{
  public interface IVolume{
    function volume():String;
  }//end interface
}//end package
```

This interface declares the method named **volume** . Because this interface is extended by the interface named **IArea** , any class that implements **IArea** inherits the abstract method named **volume** and must provide a concrete definition for the method.

The file named ICircumference.as

The interface named **ICircumference** is very similar to the interface named **IVolume** . The code for this interface is provided in Listing 14 near the end of the lesson.

This interface declares the method named **circumference** . Once again, because this interface is extended by the interface named **IArea** , any class that implements **IArea** inherits the abstract method named **circumference** and must provide a concrete definition for the method.

The file named MyCircle.as

The class named **MyCircle** is shown in its entirety in Listing 9

Example:
The class named MyCircle.

```
package CustomClasses{
  public class MyCircle implements IArea{
    private var radius:Number;

    public function MyCircle(radius:Number)
{//constructor
      this.radius = radius;
    }//end constructor

    public function area():String{
      return "Radius = " + radius + "\n" +
             "Area = " + Math.PI * radius *
radius;
    }//end area

    public function id():String{
      return "Circle\n";
    }//end id

    public function circumference():String{
      return "Radius = " + radius + "\n" +
             "Circumference = " + 2 * Math.PI *
radius;
    }//end function circumference
    public function volume():String{
      //Assumes that the shape is a cylinder with
a
      // depth of ten units.
      return "Radius = " + radius + "\n" +
             "Depth = 10\n" +
             "Volume = " + 10 * Math.PI * radius *
radius;
    }//end area

  }//end class
}//end package
```

Inherits four abstract methods

As you can see, this class implements the interface named **IArea** , causing it to inherit **the following abstract methods** :

- **area**
- **id**
- **circumference**
- **volume**

The **area** and **id** methods are declared in the **IArea** interface and are inherited directly into the **MyCircle** class.

The **circumference** and **volume** methods are declared in the interfaces named **ICircumference** and **IVolume** , both of which are extended by the **IArea** interface. Therefore, the **MyCircle** class inherits those abstract methods by way the **IArea** interface.

Concrete definitions for four methods

Listing 9 provides concrete versions of the four inherited abstract methods. As a practical matter, this amounts to overriding inherited abstract methods. Note however, that unlike the case of overriding a concrete method inherited from a class, the keyword **override** is ***not*** required when overriding an abstract method inherited from an interface.

Other than the fact that this class implements an interface and overrides inherited abstract methods, there is nothing in Listing 9 that should be new to you.

The file named MyRectangle.as

The class named **MyRectangle** is defined in Listing 16 near the end of the lesson. This class is very similar to the **MyCircle** class shown in Listing 9. As with the **MyCircle** class, it implements the **IArea** interface. Therefore, it inherits and overrides the same four abstract methods shown in the above list .

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 10 through Listing 16. Use that code to create a Flex project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the ActionScript and MXML files discussed in this lesson are provided in Listing 10 through Listing 16 below.

Example:
Listing for the file named Interface01.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<!--Illustrates polymorphism using an interface.--
>

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Example:
Listing for the file named Driver.as.

```
package CustomClasses{
  import flash.events.*;

  import mx.containers.HBox;
  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.controls.TextArea;

  public class Driver extends VBox{
    private var textArea:TextArea = new
```

```
TextArea();
    private var myShape:IArea;

    private var randomChoice:Number;
    private var radius:uint;
    private var rectWidth:uint;
    private var rectHeight:uint;

    public function Driver(){//constructor
      var label:Label = new Label();
      label.text = "Interface Polymorphism Demo";
      label.setStyle("fontSize",14);
      label.setStyle("color",0xFFFF00);
      addChild(label);

      //Put three buttons in an HBox
      var hbox:HBox = new HBox();
      addChild(hbox);

      var areaButton:Button = new Button();
      areaButton.label = "Area";
      hbox.addChild(areaButton);

      var circButton:Button = new Button();
      circButton.label = "Circumference";
      hbox.addChild(circButton);

      var volumeButton:Button = new Button();
      volumeButton.label = "Volume";
      hbox.addChild(volumeButton);

      //Put the text area below the HBox.
      textArea.width = 245;
      textArea.height = 80;
      addChild(textArea);

      //Register a click event handler on each of
```

```
the
      // buttons
      areaButton.addEventListener(

MouseEvent.CLICK,areaButtonHandler);

      circButton.addEventListener(

MouseEvent.CLICK,circButtonHandler);

      volumeButton.addEventListener(

MouseEvent.CLICK,volumeButtonHandler);
    }//end constructor

    //Local utility method for getting and saving
four
    // random values.
    private function getRandomValues():void{
      randomChoice = Math.random();
      radius = uint(10*Math.random() + 1);
      rectWidth = uint(10*Math.random() + 1);
      rectHeight = uint(10*Math.random() + 1);
    }//end getRandomValues

    //Define click event handler methods.
    private function areaButtonHandler(

event:MouseEvent):void{
      getRandomValues();

      if(randomChoice < 0.5){
        myShape = new MyCircle(radius);
      }else{
        myShape = new
MyRectangle(rectWidth,rectHeight);
      }//end else
      textArea.text = myShape.id() +
myShape.area();
    }//end areaButtonHandler

    private function circButtonHandler(

event:MouseEvent):void{
      getRandomValues();

      if(randomChoice < 0.5){
        myShape = new MyCircle(radius);
      }else{
        myShape = new
MyRectangle(rectWidth,rectHeight);
      }//end else
      textArea.text = myShape.id() +

myShape.circumference();
    }//end circButtonHandler

    private function volumeButtonHandler(

event:MouseEvent):void{
      getRandomValues();

      if(randomChoice < 0.5){
        myShape = new MyCircle(radius);
      }else{
        myShape = new
MyRectangle(rectWidth,rectHeight);
      }//end else
      textArea.text = myShape.id() +
myShape.volume();
    }//end circButtonHandler

  }//end class
}//end package
```

Example:
Listing for the file named IArea.as.

```
package CustomClasses{
  public interface IArea extends
IVolume,ICircumference{
    function area():String;
    function id():String;
  }//end interface
}//end package
```

Example:
Listing for the file named IVolume.as.

```
package CustomClasses{
  public interface IVolume{
    function volume():String;
  }//end interface
}//end package
```

Example:
Listing for the file named ICircumference.as.

```
package CustomClasses{
  public interface ICircumference{
    function circumference():String;
  }//end interface
}//end package
```

Example:
Listing for the file named MyCircle.as.

```
package CustomClasses{
  public class MyCircle implements IArea{
    private var radius:Number;

    public function MyCircle(radius:Number)
{//constructor
      this.radius = radius;
    }//end constructor

    public function area():String{
      return "Radius = " + radius + "\n" +
             "Area = " + Math.PI * radius *
radius;
    }//end area

    public function id():String{
      return "Circle\n";
    }//end id

    public function circumference():String{
      return "Radius = " + radius + "\n" +
             "Circumference = " + 2 * Math.PI *
radius;
    }//end function circumference

    public function volume():String{
      //Assumes that the shape is a cylinder with
a
      // depth of ten units.
      return "Radius = " + radius + "\n" +
             "Depth = 10\n" +
             "Volume = " + 10 * Math.PI * radius *
radius;
    }//end area
  }//end class
}//end package
```

Example:
Listing for the file named MyRectangle.as.

```
package CustomClasses{
  public class MyRectangle implements IArea{
    private var width:Number;
    private var height:Number;

    public function MyRectangle(
                width:Number,height:Number)
{//constructor
      this.width = width;
      this.height = height;
    }//end constructor

    public function area():String{
      return "Width = " + width + "\n" +
             "Height = " + height + "\n" +
             "Area = " + width * height;
    }//end area

    public function id():String{
      return "Rectangle\n";
    }//end id

    public function circumference():String{
      return "Width = " + width + "\n" +
             "Height = " + height + "\n" +
             "Circumference = " + 2 * (width +
height);
```

```
    }//end function circumference

    public function volume():String{
      //Assumes that the shape is a rectangular
solid
      // with a depth of ten units.
      return "Width = " + width + "\n" +
             "Height = " + height + "\n" +
             "Depth = 10\n" +
             "Volume = " + 10 * width * height;
    }//end area

  }//end class
}//end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Interface Polymorphism - The Big Picture
- Files:
    - ActionScript0112\ActionScript0112.htm
    - ActionScript0112\Connexions\ActionScriptXhtml0112.htm

Note: PDF disclaimer: Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

-end-

Digging Deeper into ActionScript Events
Learn how to translate a largely Flex MXML program into a largely ActionScript program. Learn that events can be subdivided into two major categories: those that involve direct user interaction and those that don't. Learn how to use the events documentation to find what you need to write event-driven ActionScript programs.

Note: Click Effects03 to run this ActionScript program. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

Note: ActionScript 3: Note that all references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a continuing series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the

emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

- Listing 13 . Event handler for the creationComplete event dispatched by the Panel.
- Listing 14 . The common MXML code.
- Listing 15 . Driver class for ActivateEvent01.
- Listing 16 . Driver class for Effects03.

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

The ActionScript 3.0 Reference for the Adobe Flash Platform is a voluminous document. Understanding ActionScript events on the basis of that document alone can be a daunting task.

There are two entry points into the documentation that can make it somewhat easier to navigate:

- The flash.events.Event class
- The addEventListener method

The flash.events.Event class

As I understand it, all possible ActionScript events are represented by subclasses of the flash.events.Event class. That class has approximately eighty subclasses, many of which are further extended into additional subclasses.

The flash.events.MouseEvent class

For example, the flash.events.MouseEvent class has eight subclasses, some of which are extended into other subclasses. Therefore, the total number of subclasses of the **Event** class may be well in excess of one hundred. This means that there may be more than one hundred different *types* of events being dispatched during the running of an ActionScript program.

Event types and subtypes

An ActionScript class represents a *type* . Therefore, each subclass of the **Event** class defines a different type of event. Many subclasses define a large number of subtypes.

For example, the **MouseEvent** class defines about seventeen subtypes ranging in alphabetical order from **CLICK** to **ROLL_OVER** , including **MOUSE_MOVE** , **MOUSE_DOWN** , **MOUSE_UP** , etc.

> *(For brevity, I will refer to the subtypes as types for the remainder of this document.)*

Hundreds of event types and hundreds of object types

Therefore, there are many hundreds of different types of events that can be dispatched in various combinations by hundreds of different types of objects during the running of an ActionScript program.

The DisplayObject class

Objects of some classes *(such as Button)* , which are subclasses of the DisplayObject class, can dispatch events as a result of direct user interaction. For example, a user can click on a button and cause a **click** event to be dispatched.

The URLLoader class

Objects of other classes *(such as URLLoader)* , which are not subclasses of the DisplayObject class, cannot be caused to dispatch an event as a result of direct user interaction. *(As far as I know, there is no way for a user to interact directly with an object of the* ***URLLoader*** *class.)*

However, an object of the **URLLoader** class can dispatch nine different types of events ranging in alphabetical order from **activate** to **securityError** .

The addEventListener methods

The documentation index for Flex Builder 3 lists eight different versions of the **addEventListener** method in classes plus one that is declared in an interface. Of the eight versions defined in classes, one is defined in the EventDispatcher class. That is the one that I will concentrate on in this lesson.

For the record, however, four of the eight versions are overridden versions of the method that is defined in the **EventDispatcher** class. The remaining three of the eight are defined in classes that extend the **Proxy** class, which I will ignore in this lesson.

The addEventListener method in the EventDispatcher class

According to the documentation, the addEventListener method that is defined in the **EventDispatcher** class " *Registers an event listener object with an* ***EventDispatcher*** *object so that the listener receives notification of an event."*

Parameters of the addEventListener method

This method requires the following five parameters, the last three of which have default values:

1. type:String
2. listener:Function
3. useCapture:Boolean = false
4. priority:int = 0
5. useWeakReference:Boolean = false

Will concentrate on the first two parameters

I will concentrate on the first two parameters in this lesson. In addition, I will refer you to Understanding the AS3 Event Flow by Jody Hall for an excellent discussion of the purpose and use of the third parameter.

The type of the event

The first parameter , which is the type of the event, usually looks something like the following in ActionScript syntax:

Event.ACTIVATE

The event-handler function

The second parameter is the name of a function or method that processes the event. This method must accept an **Event** object as its only parameter and must return void. A sample ActionScript signature for a suitable event handler method follows:

private function activateHandler(event:Event):void

Note that the actual type of the parameter may be any subclass of **Event** , such as **MouseEvent** for example. Note also that only the name of the function *(without parentheses)* is passed to the **addEventListener** method.

Available to all subclasses of EventDispatcher

The **addEventListener** method is defined in the **EventDispatcher** class and inherited by all subclasses of that class. Therefore, the method can be called on any object instantiated from any subclass of the **EventDispatcher** class.

There are many subclasses of the EventDispatcher class

The **EventDispatcher** class has approximately seventy-five immediate subclasses that ultimately fan out to include hundreds and possibly thousands of individual classes. *(For example, the **Button** class is a* *subclass of the **EventDispatcher** class about seven levels down the inheritance hierarchy.)*

Many combinations are nonsensical

This means that hundreds of different event types can be registered on hundreds of different object types. However, many of those combinations of event types and object types make absolutely no sense at all.

The SolidColor class

For example, one of the subclasses of the **EventDispatcher** class is named **SolidColor** . You can register a handler for a **MouseEvent.CLICK** event on a **SolidColor** object with no obvious ill effects. There is no compiler error and no runtime error.

However, the registration of the event handler on the object has no effect. As far as I know, it is not possible to cause a **SolidColor** object to dispatch a **CLICK** event. Therefore, the **click** event handler will never be executed.

A weakness in the event model

In my opinion, the ability for the programmer to register event types on objects that will simply ignore events of that type is a major weakness in the ActionScript event model. Other programming languages such as Java provide more help in avoiding such programming errors.

Guarding against nonsensical combinations

How can you determine which combinations of classes and event types make sense and which do not?

As usual, your best friend is the documentation. For example, the ActionScript 3.0 Reference for the Adobe Flash Platform allows you to click on a class name in the leftmost frame and read about that class in the rightmost frame. For every class that supports events, there is a hyperlink at the top of the rightmost frame labeled **Events** . Clicking on that hyperlink will expose all of the event types supported by objects of the class being viewed.

SolidColor events

For example, objects of the **SolidColor** class support only two types of events:

- flash.events.Event.ACTIVATE
- flash.events.Event.DEACTIVATE

Although you may be able to register event handlers for other types of events on an object of the **SolidColor** class, it doesn't make any sense to do so. The documentation tells us that there is no point in registering a **MouseEvent.CLICK** event on a **SolidColor** object.

The ACTIVATE and DEACTIVATE event types

Both of these event types are defined in the **EventDispatcher** class and are inherited by all classes whose objects have the ability to dispatch events *(other than subclasses of the **Proxy** class that I am ignoring in this lesson)* . They are both dispatched by the system as the result of certain runtime conditions that may be beyond the direct control of the user. I will explain a sample program later that uses these two event types.

The Flex Builder IDE is also helpful

Other useful tools for avoiding nonsensical combinations of classes and event types are the Flex Builder 3 and Flash Builder 4 IDEs. The IDEs provides popup hints at various points as you type ActionScript code. In some cases, the popup hints will list the types of events supported by the object on which you are registering an event listener.

The DisplayObject class

Of the large number of immediate subclasses of the EventDispatcher class, the one that will probably garner most of your attention in your early

ActionScript programming experience is the class named DisplayObject . According to the documentation:

> *"The DisplayObject class is the base class for all objects that can be placed on the display list. The display list manages all objects displayed in Flash Player or Adobe AIR."*

Flex components

All of the objects with which the user can have direct interaction are instantiated from subclasses of this class. For example, I believe that all of the components that appear in the **Components** tab in the **Design** view of Flex Builder 3 or Flash Builder 4 are subclasses of the **DisplayObject** class.

I also believe that all of those classes are grouped into packages such as the following:

- **mx.controls**
- **mx.containers**
- **mx.modules**
- **mx.charts**

There are about sixty-five classes in the **Components** tab of Flex Builder 3, and those are the classes that usually involve direct user interaction. That leaves many more classes that support events that don't usually involve direct user interaction.

Events that don't involve direct user interaction

Even some of the classes that are subclasses of **DisplayObject** support events that don't involve direct user interaction such as the **activate** and **deactivate** events.

Events that don't involve user interaction are usually events that are dispatched because of some change of state within the program. For example, it is possible to register event listeners on object properties and cause other objects to be notified when the value of such properties change.

Preview

I will present and explain two programs in the remainder of this lesson. The first program named **ActivateEvent01** provides a relatively simple illustration of servicing events that are dispatched independently of direct user interaction.

The second program named **Effects03** is somewhat more substantial. It illustrates the servicing of events that are dispatched as a result of direct user interaction as well as events that are dispatched independently of direct user interaction.

Discussion and sample code

A simple MXML file

Both of the programs that I will explain in this lesson use the same simple MXML file shown in Listing 1 and also in Listing 14.

Example:
Common code for the MXML file.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">
  <cc:Driver/>
</mx:Application>
```

This MXML code simply instantiates a new object of the **Driver** class in the **cc** namespace. Beyond that point, all program behavior is controlled by ActionScript code.

The program named ActivateEvent01

This is probably the most fundamental of all event-driven ActionScript programs. This program illustrates the **activate** and **deactivate** events. According to the EventDispatcher documentation, the **activate** event is *"Dispatched when the Flash Player or AIR application gains operating system focus and becomes active."*

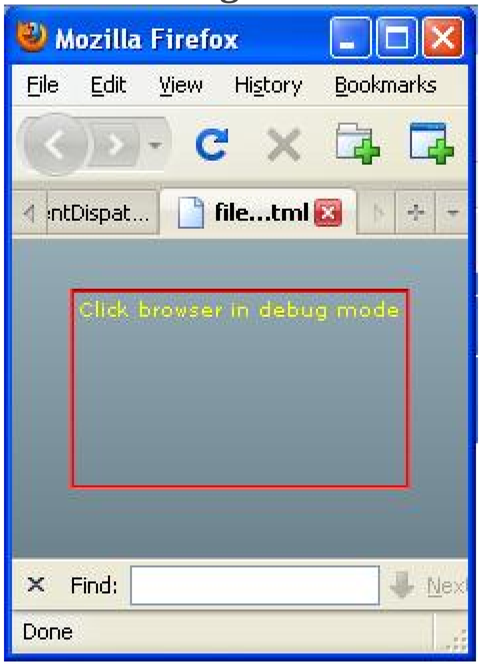
Similarly, the **deactivate** event is *"Dispatched when the Flash Player or AIR application loses the operating system focus and is becoming inactive."*

Program screen graphics

The screen output for this program is shown in Figure 1. This output doesn't change during the running of the program.

Program ActivateEvent01 screen graphics.

Figure



```
Click browser in debug mode
```

Program ActivateEvent01 screen graphics.

Must run in debug mode

This program uses calls to the **trace** function to produce output text on the console screen. Therefore, you will need to compile and run the program in debug mode in the IDE to get matching results.

Gain and then lose operating system focus

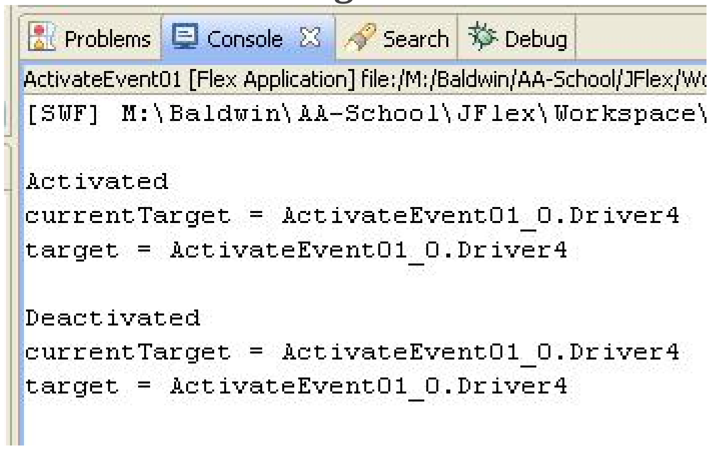
If you start the program in debug mode, click somewhere inside the Flash window and then click somewhere in another window or on the desktop, you will cause the Flash Player to first gain and then lose the operating system focus. In other words, on the first click inside the Flash window, the Flash Player will become the active program. On the second click in another window, another program will become the active program.

Debug output in the system console

When you do that, output similar to that shown in Figure 2 should appear in the IDE console.

Debug output in the system console.

Figure



ActivateEvent01 [Flex Application] file:/M:/Baldwin/AA-School/JFlex/W

```
[SWF] M:\Baldwin\AA-School\JFlex\Workspace\

Activated
currentTarget = ActivateEvent01_0.Driver4
target = ActivateEvent01_0.Driver4

Deactivated
currentTarget = ActivateEvent01_0.Driver4
target = ActivateEvent01_0.Driver4
```

Debug output in the system console.

A snapshot of the Flex Builder 3 console

Figure 2 shows a snapshot of the console window in the Flex Builder 3 IDE when the above procedure was executed in debug mode.

The three lines of text beginning with the word *Activated* were produced when the Flash Player gained operating system focus. The three lines of text beginning with the word *Deactivated* were produced when the Flash Player lost operating system focus.

The currentTarget and target properties of the Event object

For an explanation of the **currentTarget** and the **target** properties shown in Figure 2, see the excellent article titled Understanding the AS3 Event Flow by Jody Hall. As you will see in the next sample program, this information can be used to identify the component that dispatched the event.

Will discuss in fragments

I will explain the code for **ActivateEvent01** in fragments. A complete listing of the class file is provided in Listing 15 near the end of the lesson.

Beginning of Driver class for ActivateEvent01

Listing 2 shows the beginning of the **Driver** class for **ActivateEvent01** .

Example:
Beginning of Driver class for ActivateEvent01.

```
package CustomClasses{
  import flash.events.Event;
  import mx.containers.VBox;
  import mx.controls.Label;

  public class Driver extends VBox{
    public function Driver(){
      setStyle("borderStyle","inset");
      setStyle("borderColor",0xFF0000);
      height=100;

      var label:Label = new Label();
      label.text = "Click browser in debug mode"
      label.setStyle("color",0xFFFF00);
      addChild(label);


addEventListener(Event.ACTIVATE,activateHandler);

      addEventListener(Event.DEACTIVATE,

deActivateHandler);
    }//end constructor
```

Extends the VBox class

The entire program is written in the **Driver** class and illustrates the **activate** and **deactivate** events of the **EventDispatcher** class.

This class extends the **VBox** class, so an object of this class is a **VBox** object.

Register event listeners

The first statement beginning with **addEventListener** registers an event listener on the **VBox** object causing it to execute the method named

activateHandler when the Flash Player gains operating system focus and becomes the active program.

The second statement beginning with **addEventListener** registers an event listener on the **VBox** causing it to execute the **deActivateHandler** method when the Flash Player loses operating system focus and is no longer the active program.

The event handler methods

The two event handler methods are shown in Listing 3.

Example:
The event handler methods.

```
    private function 
activateHandler(event:Event):void{
        trace("\nActivated\ncurrentTarget = "
              + event.currentTarget
              + "\ntarget = "
              + event.target);
    }//end activateHandler
    //----------------------------=--------------
-----//

    private function 
deActivateHandler(event:Event):void{
        trace("\nDeactivated\ncurrentTarget = "
              + event.currentTarget
              + "\ntarget = "
              + event.target);
    }//end deActivateHandler

  }//end class
}//end package
```

Output produced by the trace function

When the methods are executed in debug mode, the call to the **trace** function extracts and displays property values from the incoming **Event** object.

This causes the material shown in Figure 2 to be displayed in the system console window each time the Flash Player gains and then loses the operating system focus.

Listing 3 also signals the end of the class and the end of the program named **ActivateEvent01** .

The program named Effects03

This is a much more substantial program, which is based on a program in the Flex documentation . That program is written mostly in Flex MXML and contains the minimum amount of ActionScript code necessary to provide the desired behavior.

Online versions of the two programs

An online executable version of the program is available following the source code listing at the above site . I encourage you to run it and observe its behavior.

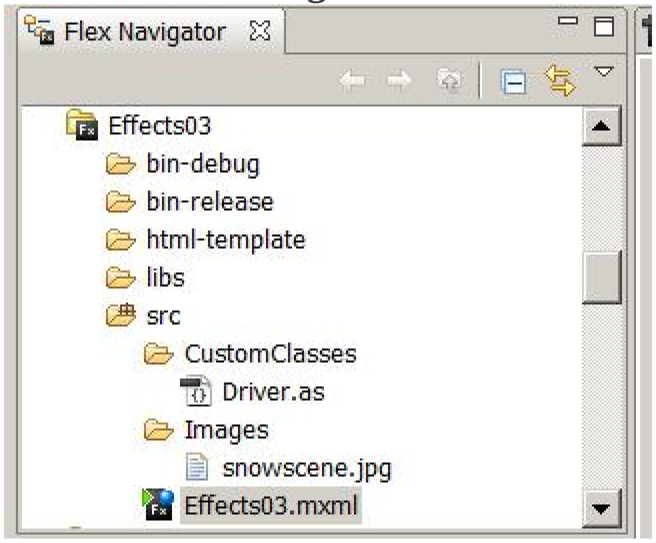
This version of the program is written almost entirely in ActionScript and contains only enough Flex MXML to make it possible to launch the program from within an HTML document. I also encourage you to run this version and observe its behavior.

Project file structure for Effects03

The project file structure for the program is shown in Figure 3, which shows a snapshot of the Flex Builder 3 Navigator panel.

Project file structure for Effects03.

Figure



```
Effects03
  bin-debug
  bin-release
  html-template
  libs
  src
    CustomClasses
      Driver.as
    Images
      snowscene.jpg
    Effects03.mxml
```

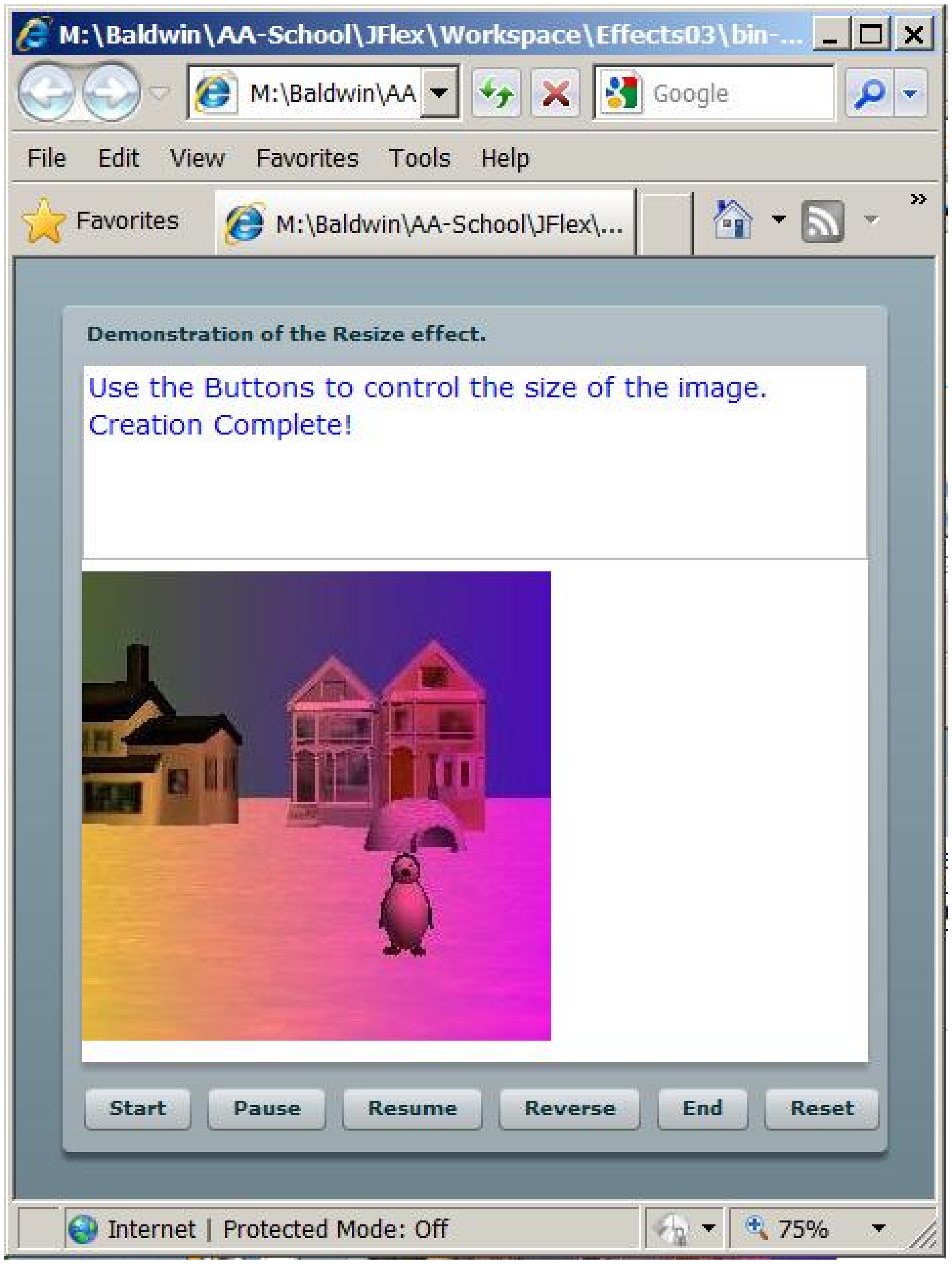
Project file structure for Effects03.

Screen display at startup

The screen display at startup is shown in Figure 4.

Screen display at startup.

Figure


M:\Baldwin\AA-School\JFlex\Workspace\Effects03\bin-...
M:\Baldwin\AA
Google
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JFlex\...
Start
Pause
Resume
Reverse
End
Reset
Internet | Protected Mode: Off
75%

Demonstration of the Resize effect.

Use the Buttons to control the size of the image.
Creation Complete!

Screen display at startup.

A creationComplete event

By the time that the snapshot shown in Figure 4 was taken, the application had already dispatched a **creationComplete** event and the event handler registered to listen for that event had caused the text "Creation Complete!" to be displayed in a **TextArea** object near the top of the browser window.

What is a creationComplete event?

According to the documentation , this event is dispatched *"when the component has finished its construction, property processing, measuring, layout, and drawing."* Therefore, other than the fact that the user starts the program running, the dispatching of this event type is completely beyond the control of the user.

Behavior of the program

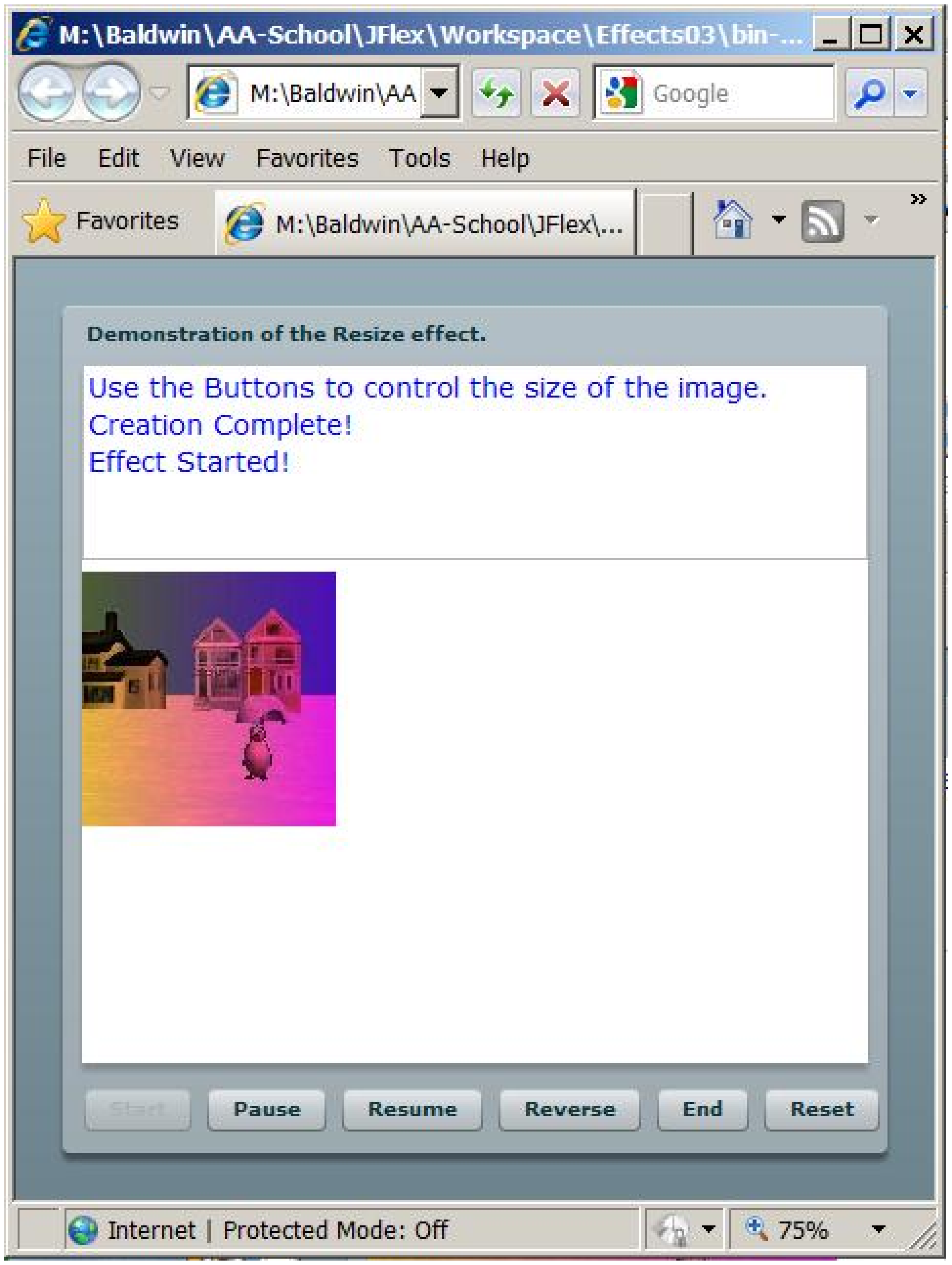
This program uses a Resize object from the mx.effects package to cause the size of the image of the penguin shown in Figure 4 to be reduced to one-fourth of its original size over a period of 10 seconds when the user clicks the **Start** button shown at the bottom of Figure 4.

The behavior of the effect resulting from clicks on the other buttons at the bottom of Figure 4 generally matches the labels on the buttons.

Screen display after clicking Start and Pause

Figure 5 shows the screen display after clicking the **Start** button, allowing the effect to run for a few seconds, and then clicking the **Pause** button.
Screen display after clicking Start and Pause.

Figure


M:\Baldwin\AA-School\JFlex\Workspace\Effects03\bin-...
M:\Baldwin\AA
Google
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JFlex\...
Demonstration of the Resize effect.
Start
Pause
Resume
Reverse
End
Reset
Internet | Protected Mode: Off
75%

```
Use the Buttons to control the size of the image.
Creation Complete!
Effect Started!
```

Screen display after clicking Start and Pause.

Servicing an effectStart event

As you will see later, clicking the **Start** button causes a method named **play** to be called on an object of the **Resize** class.

The **Resize** object dispatches an **effectStart** event whenever it starts playing an effect. An event handler, registered on the **Resize** object, causes the text **"Effect Started!"** to be displayed in the text area near the top of Figure 5.

Doesn't depend on user interaction

Even though the user clicked the **Start** button to cause the **play** method to be called on the **Resize** object in this case, the dispatching of the **effectStart** event is generally independent of user interaction and is triggered by a change of state within the program.

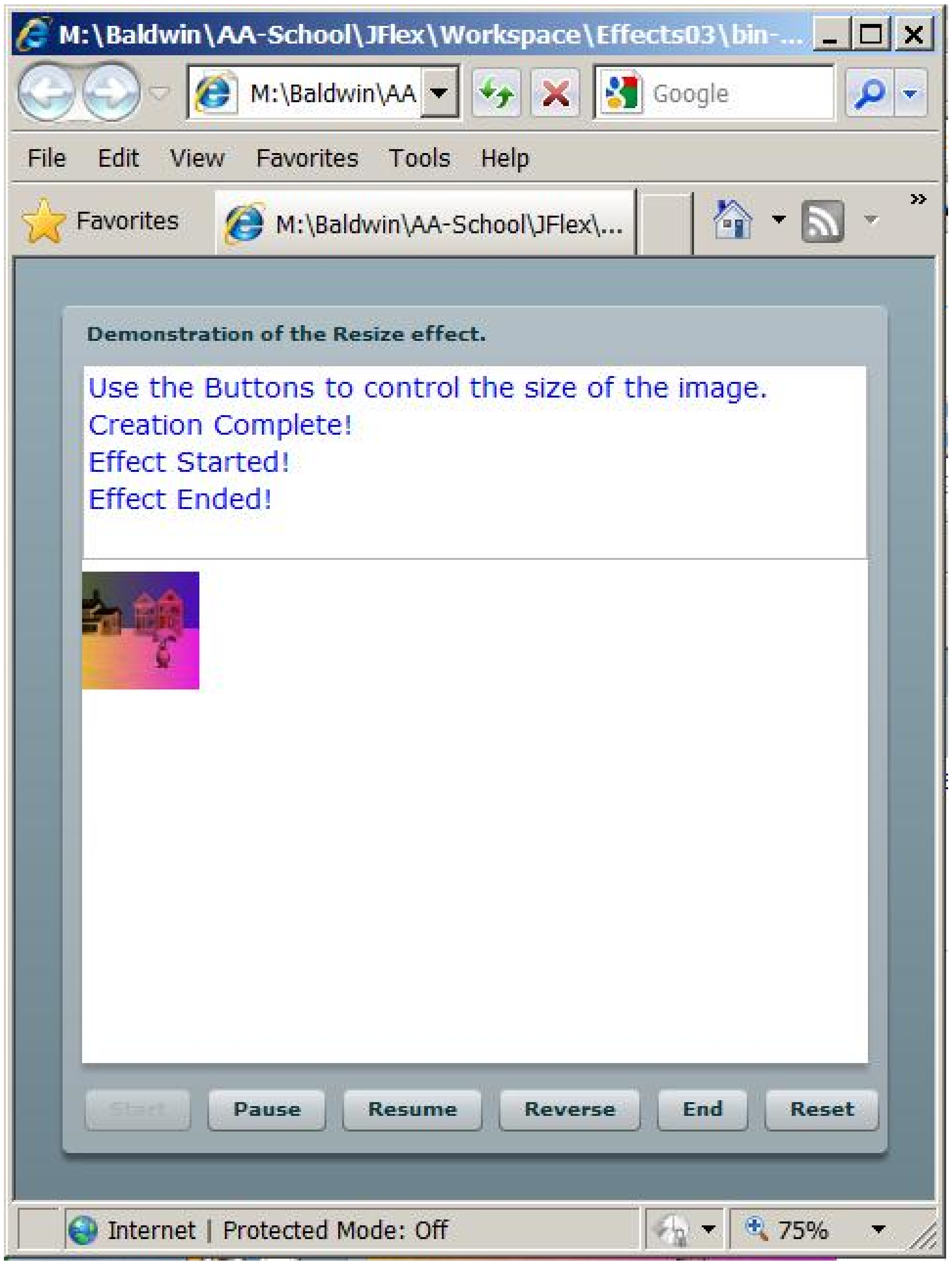
Could play the effect based on the time of day

For example, the **play** method could just as well have been called on the **Resize** object by another event handler that has determined that the time on the system clock has just struck midnight. In that case, the user could be home fast asleep and the **effectStart** event would still be dispatched.

Screen display after the effect has run to completion

The screen display in Figure 6 was captured after the user clicked the **Resume** button and allowed the effect to run to completion.
Screen display after the effect has run to completion.

Figure


M:\Baldwin\AA-School\JFlex\Workspace\Effects03\bin-...
M:\Baldwin\AA
Google
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JFlex\...
Demonstration of the Resize effect.
Start
Pause
Resume
Reverse
End
Reset
Internet | Protected Mode: Off
75%

Use the Buttons to control the size of the image.
Creation Complete!
Effect Started!
Effect Ended!

Screen display after the effect has run to completion.

Servicing an effectEnd event

The **Resize** object dispatches an **effectEnd** event when it finishes playing an effect. An event handler, registered on the **Resize** object, caused the text

"Effect Ended!" to be displayed in the text area near the top of Figure 5.

Doesn't depend on user interaction

As before, the dispatching of the **effectEnd** event is generally independent of user interaction. Instead, it is triggered by a change of state within the program. In this case, at least ten seconds has elapsed since the user clicked the **Start** button to cause the effect to be played. Once again, the **play** method could just as well have been called on the **Resize** object by another event handler that has determined that the time on the system clock has struck midnight.

The MXML file

The MXML file used to launch this program as a Flex application is shown in Listing 14. The MXML code in Listing 14 simply instantiates an object of the **Driver** class file shown in the **CustomClasses** folder in Figure 2. From that point forward, the behavior of the program is entirely controlled by the ActionScript code that begins in Listing 4.

Event-driven aspects

This program may be interesting to you from two different perspectives. First, the event-driven aspects of the program illustrate the earlier discussion of the different types of events.

Some of the events that are serviced by the program, *(such as **click** events on the buttons)* , are dispatched as the direct result of user interaction. The remainder of the events, such as a **creationComplete** event, are dispatched independently of direct user interaction.

Correspondence between ActionScript and MXML code

The second interesting aspect of the program is that it illustrates that many things can be done using either Flex MXML code or ActionScript code.

The overall structure of this program is not identical to the structure of the program shown in the documentation . However, it is similar enough that you should be able to map the MXML code from the program in the documentation to the ActionScript code in this program and understand how the two relate to one another.

Will explain in fragments

I will break this program down and explain it in fragments. Aside from the MXML file shown in Listing 14, the program consists of a single class named **Driver** . A complete listing of the **Driver** class file is provided in Listing 16 near the end of the lesson.

Beginning of the class named Driver

The Driver class begins in Listing 4. Note that this class extends the **Panel** class, causing this program to have the same overall appearance as the program provided in the documentation .

Example:
Beginning of Driver class for Effects03.

```
package CustomClasses{
  import flash.events.MouseEvent;
  import mx.containers.ControlBar;
  import mx.containers.Panel;
  import mx.controls.Button;
  import mx.controls.Image;
  import mx.controls.TextArea;
  import mx.effects.Resize;
  import mx.events.EffectEvent;
  import mx.events.FlexEvent;


  public class Driver extends Panel{
    //Instantiate and save references to all of
the
    // objects required by the program.
    private var resize:Resize = new Resize();
    private var textOut:TextArea = new TextArea();
    private var image:Image = new Image();
    private var bar:ControlBar = new ControlBar();
    private var startButton:Button = new Button();
    private var pauseButton:Button = new Button();
    private var resumeButton:Button = new
Button();
    private var reverseButton:Button = new
Button();
    private var endButton:Button = new Button();
    private var resetButton:Button = new Button();
```

In addition to containing all of the required import directives, the code in Listing 4 instantiates and saves references to all of the objects required by the program.

Beginning of the constructor for Effects03

The constructor begins in Listing 5.

Example:
Beginning of the constructor for Effects03.

```
    public function Driver(){//constructor
      this.title="Demonstration of the Resize
effect.";
      this.percentWidth = 100;
      this.percentHeight = 100;


this.addEventListener(FlexEvent.CREATION_COMPLETE,
creationCompleteHandler);
```

The code in Listing 5 sets several properties on the **Panel** object, which is the object with the light gray background shown in Figure 4. The **Panel** object is displayed in the browser window, which has a slightly darker gray background color in Figure 4.

Register an event listener

The last statement in Listing 5 registers an event listener named **creationCompleteHandler** to service creationComplete events dispatched by the **Panel** object.

As you will see later, the event handler causes the text **"Creation Complete!"** to be displayed in a **TextArea** object near the top of the **Panel** in Figure 4 *when the **Panel** object and all of its children have been constructed, initialized, and drawn.*

Prepare the TextArea object and add it to the Panel

Listing 6 sets some properties and some styles on the **TextArea** object and adds it near the top of the **Panel** as shown in Figure 4.

Example:
Prepare the TextArea object and add it to the Panel.

```
      textOut.percentWidth = 100;//percent
      textOut.height = 100;//pixels
      textOut.setStyle("color","0x0000FF");
      textOut.setStyle("fontSize",14);
      textOut.text = "Use the Buttons to control "
                            + "the size of the
```

```
image.";
      addChild(textOut);
```

This object will be used to display messages that track the progress of the program as the user clicks the buttons at the bottom of Figure 4. If the text area becomes full, a vertical scroll bar will automatically appear on the right side of the text area.

Prepare an embedded image and add it to the Panel

Listing 7 prepares an embedded image, loads it into an Image object, and adds the Image object to the **Panel** .

Example:
Prepare an embedded image and add it to the Panel.

```
      [Embed("/Images/snowscene.jpg")]
      var img:Class;
      image.load(img);

      addChild(image);
```

Comparable MXML code

In case you're interested, the ActionScript code in Listing 7 is essentially comparable to the MXML code in Figure 7.
Comparable MXML code.
Figure

```
<mx:Image id="img"
source="@Embed(
source='/Images/snowscene.jpg')"/>
```

Comparable MXML code.

Note, however, that in order to cause Figure 7 to fit into this narrow publication format, I inserted an extra line break character ahead of the second occurrence of the word "source." An MXML parser may not be willing to accept this line break.

Add the button bar to the panel

The six buttons at the bottom of Figure 4 are contained in an object of the class ControlBar , which was instantiated in Listing 4.

Listing 8 adds the **ControlBar** container to the **Panel** object.

Example:
Add the button bar to the panel.

```
      addChild(bar);
```

Prepare the six buttons for use

The code in Listing 9:

- Adds labels to each of the six buttons at the bottom of Figure 4.
- Registers the same event handler to service **click** events on each of the six buttons.
- Adds each of the six buttons to the button bar shown at the bottom of Figure 4.

Example:
Prepare the six buttons for use.

```
      //Set text on the six buttons.
      startButton.label = "Start";
      pauseButton.label = "Pause";
      resumeButton.label = "Resume";
      reverseButton.label = "Reverse";
      endButton.label = "End";
      resetButton.label = "Reset";

      //Register a click listener on each button
      startButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      pauseButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      resumeButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      reverseButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      endButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      resetButton.addEventListener(

MouseEvent.CLICK,btnHandler);

      //Add the six buttons to the button bar
      bar.addChild(startButton);
      bar.addChild(pauseButton);
      bar.addChild(resumeButton);
      bar.addChild(reverseButton);
      bar.addChild(endButton);
      bar.addChild(resetButton);
```

Configure the Resize object

The resize effect that I described earlier is accomplished by calling the **play** method on the object of the class **Resize** that was instantiated in Listing 4.

Listing 10 targets that effect to the image shown in Figure 4. Listing 10 also specifies the final size of the image and the ten-second duration during which the resize effect will play. *(Note that the original size of the image is 240x240 pixels and it will be resized to one-fourth of its original size.)*

Example:
Configure the Resize object.

```


      resize.target = image;
      resize.widthTo = 60;
      resize.heightTo = 60;
      resize.duration = 10000;

      resize.addEventListener(

EffectEvent.EFFECT_END,endEffectHandler);
      resize.addEventListener(

EffectEvent.EFFECT_START,startEffectHandler);

    } //end constructor
```

Register two event listeners

The first four statements in Listing 10 would be sufficient to play the effect if that was all that I wanted to do. In addition, however, my objective is to illustrate the servicing of events that are dispatched due to changes of state within the program.

Seven different types of events

A **Resize** object dispatches seven different types of events including the following two:

- effectStart - Dispatched when the effect starts playing.
- effectEnd - Dispatched when the effect finishes playing, either when the effect finishes playing or when the effect has been interrupted by a call to the **end()** method.

The last two statements in Listing 10 register event listeners on the **Resize** object for both of those types of events.

Listing 10 also signals the end of the constructor for the **Driver** class.

Event handlers registered on the Resize object

The two event handlers registered on the **Resize** object by Listing 10 are shown in Listing 11.

Example:
Event handlers registered on the Resize object.

```
    private function startEffectHandler(

event:EffectEvent):void{
        textOut.text += "\nEffect Started!";
    } //end event handler
    //---------------------------------------------
```

```
------//


    private function endEffectHandler(

event:EffectEvent):void{
        textOut.text += "\nEffect Ended!";
    } //end event handler
```

The first method named **startEffectHandler** is executed each time the **Resize** object dispatches an **effectStart** event. The second method named **endEffectHandler** is executed each time the **Resize** object dispatches an **effectEnd** event.

Output text at the start and the end of the effect

The text in the text area at the top of Figure 5 shows the result of executing the **startEffectHandler** method from Listing 11.

The text in the text area at the top of Figure 6 shows the result of executing the **endEffectHandler** method from Listing 11.

> *(Note the small size of the image at the end of the resize effect in Figure 6 as compared to the size of the image at the beginning of the resize effect in Figure 4.)*

Methods of the Resize class

The **Resize** class defines several methods including the following six *(in alphabetical order)* :

- **end** - Interrupts an effect that is currently playing, and jumps immediately to the end of the effect.
- **pause** - Pauses the effect until you call the **resume** method.
- **play** - Begins playing the effect.
- **resume** - Resumes the effect after it has been paused by a call to the **pause** method.
- **reverse** - Plays the effect in reverse *(if the effect is currently playing)* , starting from the current position of the effect.
- **stop** - Stops the effect, leaving the effect targets in their current state.

Common event handler for the buttons

Listing 12 shows a common event handler that is used to service **click** events on all six of the buttons at the bottom of Figure 4.

Example:
Common event handler for the buttons.

```
    private function 
btnHandler(event:MouseEvent):void{
      if (event.target == startButton) {
        resize.play();//start the effect
        startButton.enabled = false;
      }else if(event.target == pauseButton){
        resize.pause();//pause the effect
      }else if(event.target == resumeButton){
        resize.resume();//resume the effect after 
a pause
      }else if(event.target == reverseButton){
        resize.reverse();//reverse the effect
      }else if(event.target == endButton){
        resize.end();//end the effect prematurely
      }else{//reset the program to starting 
conditions
        resize.end();
        image.width=240;
        image.height=240;
        startButton.enabled=true;
      } //end else

    } //end btnHandler
```

Identify the button that dispatched the event

Listing 12 extracts and uses the **target** property of the incoming **MouseEvent** object to identify which of the six buttons dispatched the **click** event that caused the method to be executed.

Call a corresponding method on the Resize object

For each of the first five buttons shown in Figure 4, Listing 12 calls a corresponding method from the above list on the **Resize** object. *(Note that clicking the **Start** button causes the **Start** button to be disabled by the code in Listing 12. Note also that none of the buttons call the **stop** method from the above list.)*

Service the Reset button

The sixth button, labeled **Reset** also calls the **end** method from the above list to cause the effect to immediately jump to the end. Then it executes some additional code to restore the image to its original size and to re-enable the **Start** button.

Service the creationComplete event dispatched by the Panel

That brings us to the final event handler method and the end of the program. The method shown in Listing 13 is executed when the **Panel** dispatches a **creationComplete** event.

Example:
Event handler for the creationComplete event dispatched by the Panel.

```
    private function creationCompleteHandler(

event:FlexEvent):void{
        textOut.text += "\nCreation Complete!";
    } //end event handler
```

As you saw in Figure 4, the code in this method causes the text "Creation Complete!" to be displayed in the text area when the **Panel** and all of its children have been created, initialized, and drawn.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 14 through Listing 16. Use that code to create Flex projects. Compile and run the projects. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the programs discussed in this lesson are provided in Listing 14 through Listing 16.

Example:
The common MXML code.

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">
  <cc:Driver/>
</mx:Application>
```

Example:
Driver class for ActivateEvent01.

```
//Illustrates the activate and deactivate events
of the
// EventDispatcher class. Must run in debug mode
to see
// the text output.
// Click on the browser to cause the Flash player
// to gain focus and fire an activate event.
// Click on the desktop to cause the Flash player
to
// lose focus and fire a deactivate event.
package CustomClasses{
  import flash.events.Event;
  import mx.containers.VBox;
  import mx.controls.Label;

  public class Driver extends VBox{
    public function Driver(){
      setStyle("borderStyle","inset");
      setStyle("borderColor",0xFF0000);
      height=100;
      var label:Label = new Label();
      label.text = "Click browser in debug mode"
      label.setStyle("color",0xFFFF00);
      addChild(label);


addEventListener(Event.ACTIVATE,activateHandler);
      addEventListener(Event.DEACTIVATE,

deActivateHandler);
    }//end constructor


    private function
activateHandler(event:Event):void{
        trace("\nActivated\ncurrentTarget = "
              + event.currentTarget
              + "\ntarget = "
              + event.target);
    }//end activateHandler

    private function
deActivateHandler(event:Event):void{
        trace("\nDeactivated\ncurrentTarget = "
              + event.currentTarget
              + "\ntarget = "
              + event.target);
    }//end deActivateHandler

  }//end class
}//end package
```

Example:
Driver class for Effects03.

```
/*This is an ActionScript version of a program
that is
similar to the sample MXML program in the
documentation
for the Effect class at:
http://livedocs.adobe.com/flex/3/langref/mx/effect
s
/Effect.html. However, several changes were made
to the
behavior of the program to make it more suitable
for the
instructional purpose.
*/

package CustomClasses{
  import flash.events.MouseEvent;
  import mx.containers.ControlBar;
  import mx.containers.Panel;
  import mx.controls.Button;
  import mx.controls.Image;
  import mx.controls.TextArea;
  import mx.effects.Resize;
  import mx.events.EffectEvent;
  import mx.events.FlexEvent;


  public class Driver extends Panel{
    //Instantiate and save references to all of
the
    // objects needed by the program.
    private var resize:Resize = new Resize();
    private var textOut:TextArea = new TextArea();
    private var image:Image = new Image();
    private var bar:ControlBar = new ControlBar();
    private var startButton:Button = new Button();
    private var pauseButton:Button = new Button();
    private var resumeButton:Button = new
```

```
Button();
    private var reverseButton:Button = new
Button();
    private var endButton:Button = new Button();
    private var resetButton:Button = new Button();
    //----------------------------------------
------//

    public function Driver(){//constructor
      this.title="Demonstration of the Resize
effect.";
      this.percentWidth = 100;
      this.percentHeight = 100;

this.addEventListener(FlexEvent.CREATION_COMPLETE,

creationCompleteHandler);

      //Set textOut properties and add to the
panel.
      textOut.percentWidth = 100;//percent
      textOut.height = 100;//pixels
      textOut.setStyle("color","0x0000FF");
      textOut.setStyle("fontSize",14);
      textOut.text = "Use the Buttons to control "
                               + "the size of the
image.";
      addChild(textOut);

      //Prepare an embedded image and add the
Image
      // object to the panel.
      [Embed("/Images/snowscene.jpg")]
      var img:Class;
      image.load(img);
      addChild(image);
      //Add the button bar to the panel.
      addChild(bar);

      //Set text on the six buttons.
      startButton.label = "Start";
      pauseButton.label = "Pause";
      resumeButton.label = "Resume";
      reverseButton.label = "Reverse";
      endButton.label = "End";
      resetButton.label = "Reset";

      //Register a click listener on each button
      startButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      pauseButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      resumeButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      reverseButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      endButton.addEventListener(

MouseEvent.CLICK,btnHandler);
      resetButton.addEventListener(

MouseEvent.CLICK,btnHandler);

      //Add the six buttons to the button bar
      bar.addChild(startButton);
      bar.addChild(pauseButton);
      bar.addChild(resumeButton);
      bar.addChild(reverseButton);
      bar.addChild(endButton);
      bar.addChild(resetButton);


      //Configure the Resize effect. Note that the
      // original size of the image is 240x240.
      resize.target = image;
      resize.widthTo = 60;
      resize.heightTo = 60;
      resize.duration = 10000;
      resize.addEventListener(

EffectEvent.EFFECT_END,endEffectHandler);
      resize.addEventListener(

EffectEvent.EFFECT_START,startEffectHandler);

    } //end constructor
    //--------------------------------------------------
------//

    //This common button handler is used to
service click
    // event on all six of the buttons.
    private function
btnHandler(event:MouseEvent):void{
      if (event.target == startButton) {
        resize.play();//start the effect
        startButton.enabled = false;
      }else if(event.target == pauseButton){
        resize.pause();//pause the effect
      }else if(event.target == resumeButton){
        resize.resume();//resume the effect after
a pause
      }else if(event.target == reverseButton){
        resize.reverse();//reverse the effect
      }else if(event.target == endButton){
        resize.end();//end the effect prematurely
      }else{//reset the program to starting
conditions
        resize.end();
        image.width=240;
        image.height=240;
        startButton.enabled=true;
      } //end else

    } //end btnHandler
    //--------------------------------------------------
------//

    //This event handler method is executed when
the
    // effect ends.
    private function endEffectHandler(

event:EffectEvent):void{
        textOut.text += "\nEffect Ended!";
    } //end event handler
    //--------------------------------------------------
------//

    //This event handler method is executed when
the
    // effect starts
    private function startEffectHandler(

event:EffectEvent):void{
        textOut.text += "\nEffect Started!";
    } //end event handler
    //--------------------------------------------------
------//

    //This event handler method is executed when
the
    // application dispatches a creationComplete
```

```
event.
    private function creationCompleteHandler(

event:FlexEvent):void{
        textOut.text += "\nCreation Complete!";
    } //end event handler
    //----------------------------------------------
------//

  } //end class
} //end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Digging Deeper into ActionScript Events
- Files:
    - ActionScript0114\ActionScript0114.htm
    - ActionScript0114\Connexions\ActionScriptXhtml0114.htm

-end-

Handling Keyboard Events
Learn about focus and keyboard events.

Note: Click KeyboardEvent02 to run the ActionScript program from this lesson. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

Note: All references to ActionScript in this lesson are references to version 3.0 or later.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. The lesson titled **Using Flex 3 in a Flex 4 World** was added later to accommodate the release of Flash Builder 4. *(See Baldwin's Flex programming website .)* You should study those lessons before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

In the past few lessons, you have learned how to handle mouse events under a variety of circumstances. Up to this point, however, I haven't discussed much if anything about keyboard events.

Handling keyboard events differs from handling mouse events in one major respect -- focus.

What is focus?

At any instant in time, only one object in only one application of potentially many running applications can have the focus. The object that has the focus is the only object that can respond to the keyboard. Therefore, before an object can fire events of type **KeyboardEvent** , that object must have the focus.

There is more than one way to cause an object to gain the focus. I will show you one way in this lesson. Then I will show you how to handle events of type **KeyboardEvent** fired by that object.

Preview

Run the ActionScript program named KeyboardEvent02

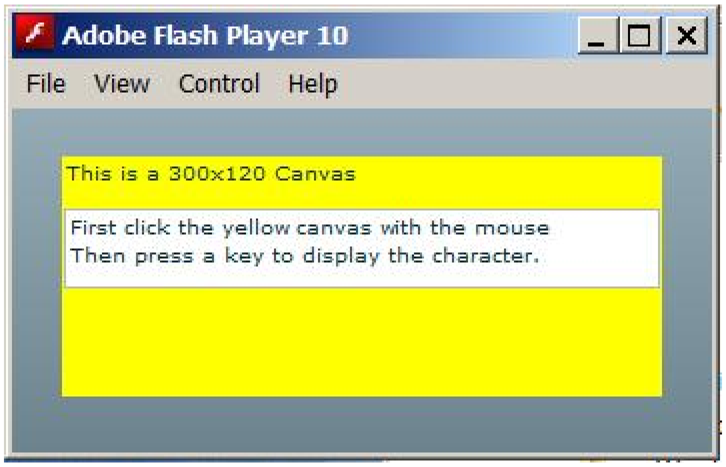
If you have the Flash Player plug-in *(version 10 or later)* installed in your browser, click here to run the program that I will explain in this lesson.

If you don't have the proper Flash Player installed, you should be notified of that fact and given an opportunity to download and install the Flash Player plug-in program.

Program output as startup

The image in Figure 1 is similar to what you should see in your browser when you start the program named **KeyboardEvent02** running.
KeyboardEvent02 output at startup.

Figure



This is a 300x120 Canvas

First click the yellow canvas with the mouse
Then press a key to display the character.

KeyboardEvent02 output at startup.

The objects

The yellow rectangle in Figure 1 is a **Canvas** object with a yellow background color.

The text in the upper-left corner is the text on a **Label** object.

The text in the white rectangle is text in an object of the class **TextArea** . This object allows the user to enter text, but that isn't the purpose of the object in this program. The purpose of the object in this program is simply to provide an output consisting of multi-line operating instructions.

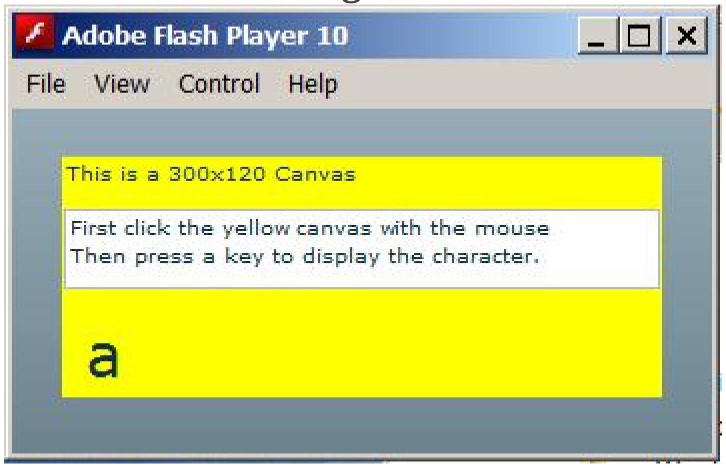
The operating instructions

At startup, the program will not respond to keyboard input. Clicking the yellow **Canvas** object with the mouse causes it to gain the focus. Once it has the focus, the canvas will respond to keyboard input.

Output after clicking the Canvas and pressing a key

Figure 2 shows the program output after clicking the yellow **Canvas** object with the mouse and then pressing the "a" key.
KeyboardEvent02 output after clicking the Canvas and pressing a key.

Figure



This is a 300x120 Canvas

First click the yellow canvas with the mouse
Then press a key to display the character.

a

KeyboardEvent02 output after clicking the Canvas and pressing a key.

The output

As you can see, this caused the letter "a" to be displayed in a large font size in the lower-left corner of the canvas. This will work for any of the letter or number keys, with or without holding down the shift key. Other keys, such as the arrow keys, don't produce a visible output however.

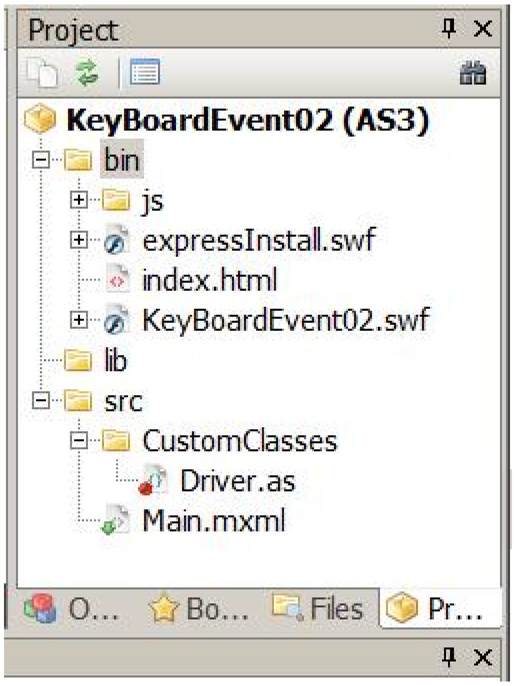
Discussion and sample code

The project file structure

The final project file structure, captured from the FlashDevelop project window, is shown in Figure 3.

Project file structure for KeyboardEvent02.

Figure



Project file structure for KeyboardEvent02.

Will explain in fragments

I will explain the code for this program in fragments. Complete listings of the MXML code and the ActionScript code are provided in Listing 6 and Listing 7 near the end of the lesson.

The MXML code

The MXML code is shown in Listing 1 and again in Listing 6 near the end of the lesson.

Example:
Code for the file named Main.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<!--
KeyboardEvent02
See explanation in the file named Driver.as
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

As is often the case in this series of lessons, the MXML file is very simple because the program was coded almost entirely in ActionScript. The MXML code simply instantiates an object of the **Driver** class. From that point forward, the behavior of the program is controlled by ActionScript code.

The ActionScript code

Beginning of the Driver class

The driver class begins in Listing 2.

Example:
Beginning of the Driver Class.

```
/*KeyboardEvent02 06/03/10
Illustrates the use of KeyboardEvent, charCode
```

```
values,
absolute positioning on a Canvas object, focus,
and a
TextArea object among other things.

See
http://livedocs.adobe.com/flex/3/langref/flash/eve
nts
/KeyboardEvent.html
**************************************************
*******/
package CustomClasses{
  import flash.events.KeyboardEvent;
  import flash.events.MouseEvent;
  import mx.containers.Canvas;
  import mx.controls.Label;
  import mx.controls.TextArea;

  public class Driver extends Canvas{
    //Instantiate and save references to the
    // objects needed by the program.
    private var instrArea:TextArea = new
TextArea();
    private var targetLabel:Label = new Label();
    private var canvasLabel:Label = new Label();
```

Extending the Canvas class

With the possible exception of the fact that the **Driver** class extends the **Canvas** class, there is nothing new in Listing 2. An object of the **Driver** class is a **Canvas** object.

I elected to extend the **Canvas** class because this makes it possible to position objects added as children of that class using absolute location coordinates.

The constructor for the Driver class

The constructor is shown in its entirety in Listing 3.

Example:
The constructor for the Driver class.

```
    public function Driver() {//constructor
      //Set the size of the Canvas object.
      this.width = 300;
      this.height = 120;

      //Prepare the TextArea and the labels.
      canvasLabel.text = "This is a 300x120
Canvas";

      instrArea.text = "First click the yellow
canvas "
          + "with the mouse\nThen press a key to
display "
          + "the character.";
      instrArea.width = 298;
      instrArea.height = 40;
      instrArea.x = 1;
      instrArea.y = 26;

      targetLabel.setStyle("fontSize", 30);
      targetLabel.x = 10;
      targetLabel.y = 78
      //Display an empty string at startup.
      targetLabel.text = "";

      //Add the labels and TextArea to the Canvas.
      this.addChild(canvasLabel);
      this.addChild(instrArea);
      this.addChild(targetLabel);
      //Set the Canvas background color to yellow.
      this.setStyle("backgroundColor",
"0xFFFF00");

      //Register two event listeners on the
canvas.
      this.addEventListener(
                          MouseEvent.CLICK,
clickHandler);
      this.addEventListener(

KeyboardEvent.KEY_DOWN,eventHandler);

    } //end constructor
```

There are several things in Listing 3 that deserve an explanation beyond the embedded comments.

Set the size

The constructor in Listing 3 begins by setting the **width** and **height** properties of the **Canvas** object to set the overall dimensions of the object in pixels.

Set the text for the upper-left Label object

The constructor sets the **text** property of the **Label** object referred to by **canvasLabel** to the small text shown in the upper-left corner of Figure 1. Since the x and y coordinate values for this object are not purposely set, they will each have a default value of zero. This will cause them to be placed in the upper-left corner of the canvas as shown in Figure 1.

Set various properties for the TextArea object

Then the constructor sets various property values for the **TextArea** object, including its text, its dimensions, and its location coordinates. As I mentioned earlier, the sole purpose of this object in this program is to provide operating instructions.

Set various properties for the target Label object

The **Label** object that is referred to by **targetLabel** in Listing 3 is used to display the character for the key that is pressed as shown by the large lower-case "a" in Figure 2.

The constructor sets various properties for this object including a font size of 30 points and an initial string value that is an empty string.

Add the objects to the Canvas

Then the constructor calls the **addChild** method on the **Canvas** object three times in succession to add the three objects to the canvas in the locations specified by their location coordinates.

Set the background color to yellow

The constructor sets the background color of the canvas to yellow. Otherwise, it would be indistinguishable from the gray background color of the Flash window.

Register event listeners

Finally, the constructor registers two event listeners on the **Canvas** object.

A MouseEvent.CLICK listener

The first event listener that is registered is one that will handle events of the type **MouseEvent.CLICK** . As you will see shortly, this handler causes the **Canvas** object to gain the focus when the user clicks the canvas with the mouse.

A KeyboardEvent.KEY_DOWN listener

The second listener that is registered is one that will handle events of the type **KeyboardEvent.KEY_DOWN** and display the character for the key that is pressed as shown by the large lower-case "a" in Figure 2.

The MouseEvent.CLICK listener

This listener is shown in its entirety in Listing 4.

Example:
The MouseEvent.CLICK listener.

```
    private function
clickHandler(event:MouseEvent):void {
      stage.focus = this;
    }//end clickHandler
```

As I explained earlier, the sole purpose of this event listener is to make it possible for the user to cause the yellow canvas object to gain the focus.

Rather than attempt to explain the one statement in the method in Listing 4, I will refer you to the page in the Adobe documentation that explains it.

The KeyboardEvent.KEY_DOWN listener

The method shown in Listing 5 is executed each time any key is pressed. However, some of the keys, such as the shift key, aren't represented by a character that can be displayed.

Example:
The KeyboardEvent.KEY_DOWN listener.

```
    private function eventHandler(

event:KeyboardEvent):void {
      targetLabel.text =

String.fromCharCode(event.charCode);
    } //end eventHandler
```

The charCode property

Each time the method is called, it receives a reference to an object of type **KeyboardEvent** . This object encapsulates several types of information about the key that was pressed. One such piece of information is a property named **charCode** . Here is some of what the documentation has to say about this property:

> *"Contains the character code value of the key pressed or released. The character code values are English keyboard values. For example, if you press Shift+3, charCode is # on a Japanese keyboard, just as it is on an English keyboard."*

The keyCode property

Another interesting property of the incoming object is the property named keyCode . This property can be used to identity any key on the keyboard, including those that are not represented by printable characters such as the shift key and the arrow keys.

Modify the text in targetLabel

The objective of the code in Listing 5 is to modify the text in the label referred to by **targetLabel** to cause it to reflect the character for the key that was pressed.

Convert charCode to a string

The **text** property of the label is type **String** . That means that the value of **charCode** must be converted to type **String** .

This is accomplished by calling the static **fromCharCode** method of the String class.

Because this is a static method, it can be called by joining its name to the name of the class to which it belongs: **String** .

Assign the returned String value to the text property

The method returns a reference to a **String** object containing the character whose **charCode** is passed as a parameter to the method. This string is then assigned to the **text** property of the target label, producing the output shown in Figure 2.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 6 and Listing 7. Use that code to create your own project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the programs discussed in this lesson are provided below.

Example:
Code for the file named Main.mxml.

```
<?xml version="1.0" encoding="utf-8"?>

<!--
KeyboardEvent02
See explanation in the file named Driver.as
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Example:
Source code for the class named Driver.

```
/*Prob04 06/03/10
Illustrates the use of KeyboardEvent, charCode
values,
absolute positioning on a Canvas object, focus,
and a
TextArea object among other things.

See
http://livedocs.adobe.com/flex/3/langref/flash/eve
nts
/KeyboardEvent.html
**************************************************
*******/
```

```
package CustomClasses{
  import flash.events.KeyboardEvent;
  import flash.events.MouseEvent;
  import mx.containers.Canvas;
  import mx.controls.Label;
  import mx.controls.TextArea;

  public class Driver extends Canvas{
    //Instantiate and save references to the
    // objects needed by the program.
    private var instrArea:TextArea = new
TextArea();
    private var targetLabel:Label = new Label();
    private var canvasLabel:Label = new Label();
    //--------------------------------------------
------//

    public function Driver() {//constructor
      //Set the size of the Canvas object.
      this.width = 300;
      this.height = 120;

      //Prepare the TextArea and the labels.
      canvasLabel.text = "This is a 300x120
Canvas";

      instrArea.text = "First click the yellow
canvas "
          + "with the mouse\nThen press a key to
display "
          + "the character.";
      instrArea.width = 298;
      instrArea.height = 40;
      instrArea.x = 1;
      instrArea.y = 26;

      targetLabel.setStyle("fontSize", 30);
      targetLabel.x = 10;
      targetLabel.y = 78
      //Display an empty string at startup.
      targetLabel.text = "";

      //Add the labels and TextArea to the Canvas.
      this.addChild(canvasLabel);
      this.addChild(instrArea);
      this.addChild(targetLabel);

      //Set the Canvas background color to yellow.
      this.setStyle("backgroundColor",
"0xFFFF00");

      //Register two event listeners on the
canvas.
      this.addEventListener(
                          MouseEvent.CLICK,
clickHandler);
      this.addEventListener(

KeyboardEvent.KEY_DOWN,eventHandler);

    } //end constructor
    //--------------------------------------------------
------//

    //This method is executed when any key is
pressed.
    // Note, however, that some keys, such as the
shift
    // key, don't have displayable charCode
values.
    private function eventHandler(

event:KeyboardEvent):void {
      targetLabel.text =
String.fromCharCode(event.charCode);
    } //end eventHandler
    //--------------------------------------------------
------//

    //This event handler is required to cause the
Canvas
    // to gain the focus and respond to keyboard
events.
    // See the URL listed earlier. (Note that
focus can
    // also be gained by pressing the tab key.)
    private function
clickHandler(event:MouseEvent):void {
      stage.focus = this;
    }//end clickHandler
    //--------------------------------------------------
------//

  } //end class
} //end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Handling Keyboard Events
- Files:
  - ActionScript0115\ActionScript0115.htm

- ActionScript0115\Connexions\ActionScriptXhtml0115.htm

-end-

Events, Triggers, and Effects
Learn about the relationships among events, triggers, and effects. Learn about two different ways to write ActionScript code to play effects. One way calls the setStyle method on the target component passing an effect trigger and an effect as parameters to the method. The other way creates an Effect object targeted to the component and calls the play method on the object. Also learn how to play multiple effects in parallel or in sequence.

Note: Click Effects04 or Effects05 to run the ActionScript programs from this lesson. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

- Miscellaneous

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Most of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

According to Using Behaviors ,

> *"Behaviors let you add animation and motion to your application in response to user or programmatic action. For example, you can use behaviors to cause a dialog box to bounce slightly when it receives focus, or to slowly fade in when it becomes visible."*

You program behaviors into your applications using MXML, ActionScript, triggers, and effects.

According to About behaviors ,

> *"A behavior is a combination of a trigger paired with an effect. A trigger is an action, such as a mouse click on a component, a component getting focus, or a component becoming visible. An effect is a visible or audible change to the target component that occurs over a period of time, measured in milliseconds. Examples of effects are fading, resizing, or moving a component."*

Triggers are not events

Triggers are caused by events, but triggers are different from events. For example, the trigger named **mouseDownEffect** results from the occurrence of a **mouseDown** event.

If an **Effect** object, such as a **Glow** effect has been associated with a **mouseDownEffect** trigger for a given target component, the **Glow** effect will be played when the user presses the mouse button while the mouse pointer is over the target component. This will be true ***regardless*** of whether or not a **mouseDown** event listener is registered on the component.

Thirteen standard triggers in Flex Builder 3

The UIComponent class **lists thirteen** triggers :

- **addedEffect** - Triggering Event: added. Played when the component is added as a child to a Container.
- **creationCompleteEffect** - Triggering Event: creationComplete Played when the component is created.
- **focusInEffect** - Triggering Event: focusIn Played when the component gains keyboard focus.
- **focusOutEffect** - Triggering Event: focusOut Played when the component loses keyboard focus.
- **hideEffect** - Triggering Event: hide Played when the component becomes invisible.
- **mouseDownEffect** - Triggering Event: mouseDown Played when the user presses the mouse button while over the component.

- **mouseUpEffect** - Triggering Event: mouseUp Played when the user releases the mouse button while over the component.
- **moveEffect** - Triggering Event: move Played when the component is moved.
- **removedEffect** - Triggering Event: removed Played when the component is removed from a Container.
- **resizeEffect** - Triggering Event: resize Played when the component is resized.
- **rollOutEffect** - Triggering Event: rollOut Played when the user rolls the mouse so it is no longer over the component.
- **rollOverEffect** - Triggering Event: rollOver Played when the user rolls the mouse over the component.
- **showEffect** - Triggering Event: show Played when the component becomes visible.

Effects are subclasses of the Effect class

Effects are subclasses of the **Effect** class a couple of levels down the inheritance hierarchy. Flex Builder 3 provides a number of built-in effects including the following:

- AnimateProperty
- Blur
- Dissolve
- Fade
- Glow
- Iris
- Move
- Pause
- Resize
- Rotate
- SoundEffect
- WipeDown
- WipeLeft
- WipeRight
- WipeUp
- Zoom

In addition, you can create your own effects.

One trigger, many effects

The same trigger can be used to trigger different types of effects. I suppose that in theory, you could create a different behavior for all possible combinations of the thirteen triggers and the sixteen different effects in the two lists provided above. In addition, you can program for multiple effects to play in response to a single trigger.

To use an effect...

By default, Flex components do not play an effect when a trigger occurs. To configure a component to use an effect, you must associate an effect with a trigger.

Preview

Two ways to play effects

There are at least two different ways to cause an effect to be played on a component in an ActionScript program. One way is to call the **setStyle** method on the component and associate an effect trigger with an effect. With that approach, the effect will be played each time the effect trigger fires.

The second way

The second way is to target an **Effect** object to the component and then call the **play** method on the effect object. This approach doesn't make explicit use of the effect trigger.

Two different programs

I will present and explain two different programs in this lesson. The first program will illustrate the first approach described above. The second program will illustrate and contrast the two approaches.

Discussion and sample code

A simple MXML file

Both programs that I will explain in this lesson are written almost entirely in ActionScript. There is just enough MXML code to make it possible to launch the programs from a browser window.

The MXML file is shown in Listing 1 and also in Listing 16.

Example:
The MXML file used for both programs.

```
<?xml version="1.0" encoding="utf-8"?>

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

As you can see, this MXML file simply instantiates an object of the class named **Driver** in the **cc** namespace. Beyond that, the entire behavior of the program is controlled by ActionScript code.

The program named Effects04

Will explain in fragments

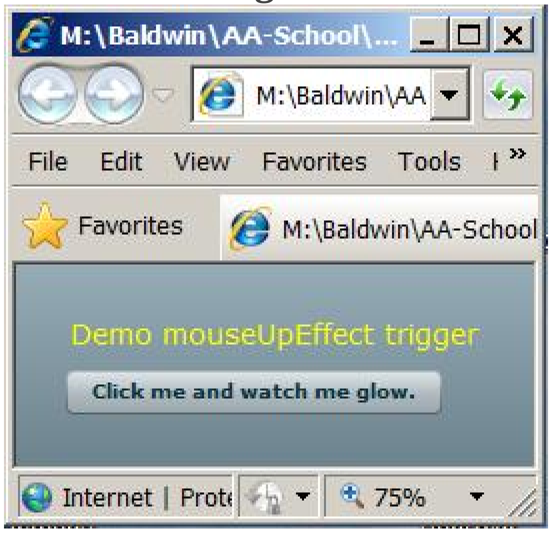
I will break the code for these two programs down and explain the code in fragments. Complete listings for the **Driver** classes for the two programs are provided in Listing 17 and Listing 18 near the end of the lesson.

Program output at startup

You can run this program online to get a better feel for its behavior. Figure 1 shows the program output at startup.

Program output at startup for Effects04.

Figure



Program output at startup for Effects04.

As you can see, the Flash Player output consists of a label and a button with the text *"Click me and watch me glow"* .

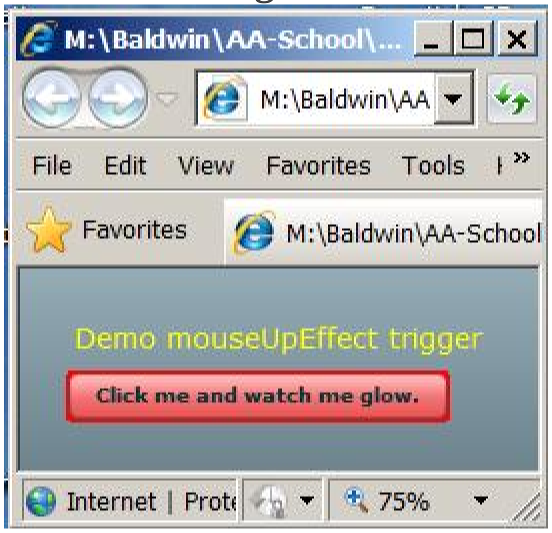
Associate a trigger with an effect

This program associates a mouseUpEffect trigger with a Glow effect to cause the button to glow when the user releases the mouse button while the mouse pointer is over the button.

Program output after clicking the button

Figure 2 shows the program output shortly after clicking the button.
Program output after clicking the button.

Figure



Demo mouseUpEffect trigger

Click me and watch me glow.

Program output after clicking the button.

As promised, the button begins to glow red when the mouse button is released while the mouse pointer is over the **Button** component. The button continues to glow for several seconds.

Beginning of the Driver class for Effects04

This program shows how to set the style on an object with a **mouseUpEffect** trigger and cause the object to glow. The entire program is written in the class named **Driver** . Listing 2 shows the beginning of the **Driver** class.

Example:
Beginning of the Driver class for Effects04.

```
package CustomClasses{

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.effects.Glow;

  public class Driver extends VBox{
    //Instantiate and save references to all of
the
    // objects needed by the program.
    private var title:Label = new Label();
    private var button:Button = new Button();
    private var glowEffect:Glow = new Glow();
```

You shouldn't find any surprises in Listing 2. I will simply highlight the last statement that instantiates an object of the **Glow** class. Note also that the **Driver** class extends the **VBox** container.

Beginning of the constructor for Effects04

The constructor begins in Listing 3.

Example:
Beginning of the constructor for Effects04.

```
    public function Driver(){//constructor

      //Set title properties and add to the VBox.
      title.setStyle("color","0xFFFF00");
      title.setStyle("fontSize",14);
      title.text = "Demo mouseUpEffect trigger";
      addChild(title);

      button.label = "Click me and watch me
glow.";
      addChild(button);
```

Once again, you shouldn't find anything new in Listing 3. This part of the constructor simply creates the yellow text label and the button shown in Figure 1 and adds them to the **VBox** .

Configuring the Glow effect

The first three statements in Listing 4 set property values on the **Glow** object that was instantiated in Listing 2.

Example:
Configuring the Glow effect.

```
      glowEffect.color = 0xFF0000;
      glowEffect.strength = 255;
      glowEffect.duration = 10000;

      button.setStyle("mouseUpEffect",glowEffect);

    } //end constructor
    //--------------------------------------------
------//

  } //end class
} //end package
```

The strength property

The purpose of the three properties should be fairly obvious on the basis of their names. However, here is what the documentation has to say about the **strength** property:

> *"The strength of the imprint or spread. The higher the value, the more color is imprinted and the stronger the contrast between the glow and the background. Valid values are from 0 to 255."*

A maximum red glow for ten seconds

The code in Listing 4 sets the effect to glow red with the maximum allowable strength and to continue to glow for ten seconds. If you run the program, you will see that the glow doesn't stay the same for ten seconds and then turn off. Instead, it decays over time.

Associate the trigger and the effect

The statement that **begins with button.setStyle** in Listing 4 is the ***key statement*** in the entire program. This statement associates the effect referenced by **glowEffect** with a **mouseUpEffect** trigger. Therefore, the effect will be played each time the mouse button is released while the mouse pointer is over the **Button** component. In fact, it doesn't even matter if the mouse pointer was over the **Button** component when the mouse button was pressed as long as it is over the **Button** component when the mouse button is released.

The end of the program

Listing 4 also signals the end of the program. Note that even though the **mouseUpEffect** trigger resulted from a **mouseUp** event, an event listener was not registered to listen for and to service the **mouseUp** event.

Three steps are required

Generally speaking, three steps are required to implement this approach:

1. Instantiate an object of the desired effect from the above list , or from a custom effect if you have one.
2. Set property values on the effect object to cause it to have the desired behavior as in Listing 4.
3. Call the **setStyle** method to associate the effect with an effect trigger from the above list as in Listing 4.

That is all that is required to play an effect when an effect trigger occurs.

The program named Effects05

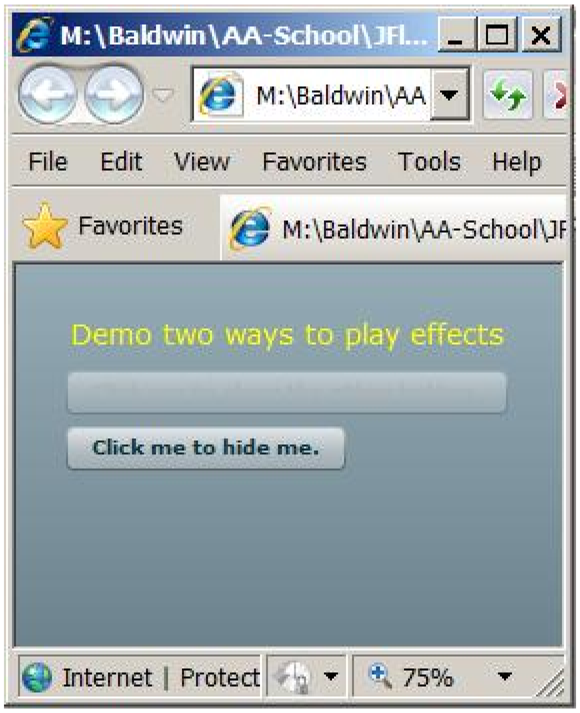
Suppose you want to do something a little more creative, such as to cause the effect that is played for a particular trigger to differ from one time to the next depending on some condition in the program.

Or, perhaps you want to play an effect on a component completely independent of triggers, such as when a player's score in a game reaches 10,000. I will show you how to do those kinds of things in this program.

Program output at startup

The best thing that you could do at this point would be to run the program online. That way, you can interact with the program as you read the following.

Figure 3 shows the program output at startup.
Program output at startup for Effects05.
Figure

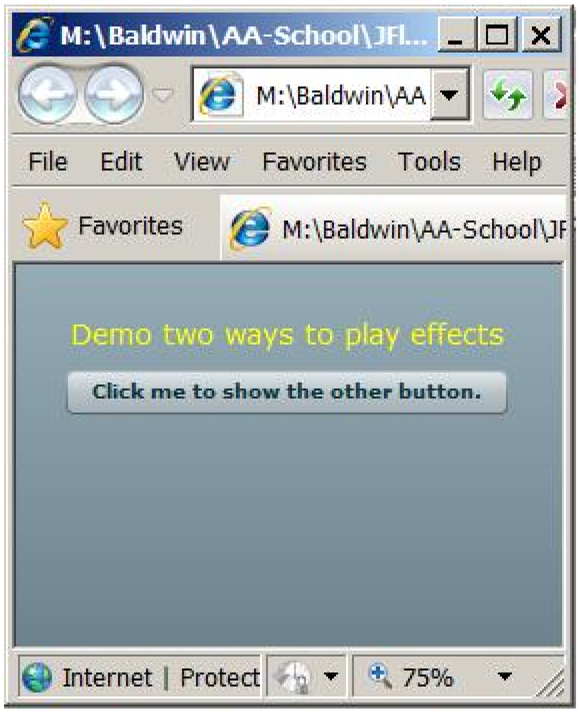

M:\Baldwin\AA-School\JFl...
M:\Baldwin\AA
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JF
Demo two ways to play effects
Click me to hide me.
Internet | Protect
75%

Program output at startup for Effects05.

At this point, the output consists of one label and two buttons. The top button is disabled and the bottom button is asking to be clicked in order to be hidden.

Program output after clicking the bottom button

Figure 4 shows the program output after the bottom button from Figure 3 has been clicked.
Program output after clicking the bottom button.
Figure


M:\Baldwin\AA-School\JFl...
M:\Baldwin\AA
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JF
Demo two ways to play effects
Click me to show the other button.
Internet | Protect
75%

Program output after clicking the bottom button.

Only the top button is showing

At this point, the bottom button in Figure 3 has been hidden and the top button in Figure 3 has been enabled. From this point forward, the user will alternate between clicking the top and bottom buttons.

Do it several times

You need to go through the sequence several times to experience the full effect. Each time the user clicks the top button, it becomes disabled and the bottom button becomes visible. Each time the bottom button becomes visible, an effect is played. Effects are played in the following sequence:

1. A WipeRight effect.
2. A Rotate effect.
3. A Glow effect.

4. All three of the above in parallel.

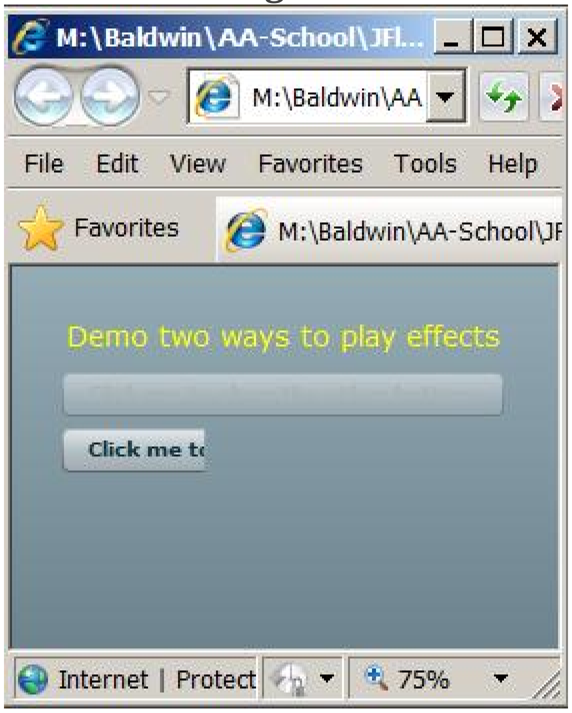
The sequence repeats after the three effects are played in parallel.

The next four figures show screen shots of the effects listed above caught in midstream.

The WipeRight effect

Figure 5 shows the restoration of the bottom button with a **WipeRight** effect. As you can see, only part of the button was visible when the screen show was taken.
The WipeRight effect.

Figure


M:\Baldwin\AA-School\JFl...
M:\Baldwin\AA
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JF
Demo two ways to play effects
Click me t
Internet | Protect
75%

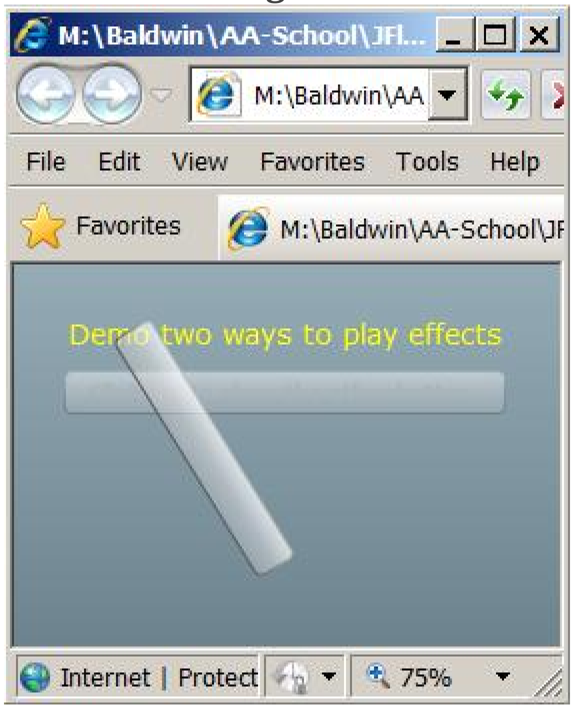
The WipeRight effect.

The Rotate effect

Figure 6 shows the **Rotate** effect caught in midstream.

The Rotate effect.

Figure


M:\Baldwin\AA-School\JFl...
M:\Baldwin\AA
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JF
Demo two ways to play effects
Internet | Protect
75%

The Rotate effect.

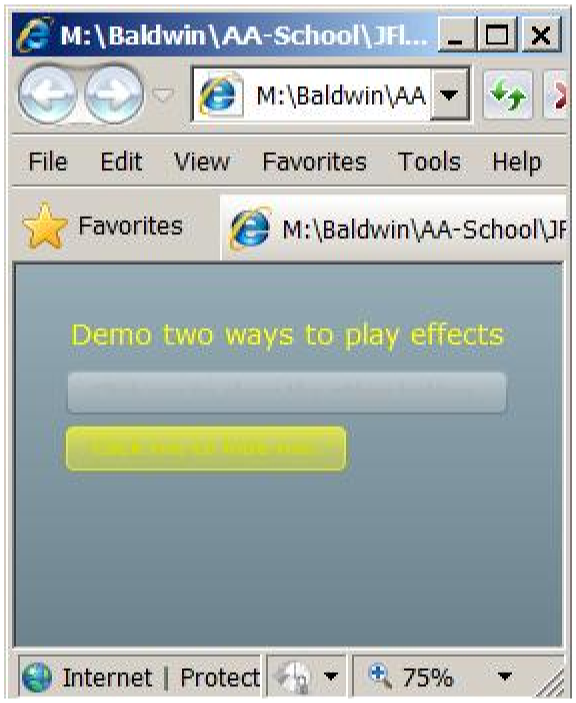
The bottom button rotates a full 360 degrees around its center point before coming to rest in the position shown in Figure 3 with its label restored.

The Glow effect

Figure 7 shows the bottom button in the middle of a yellow glow effect.

The Glow effect.

Figure

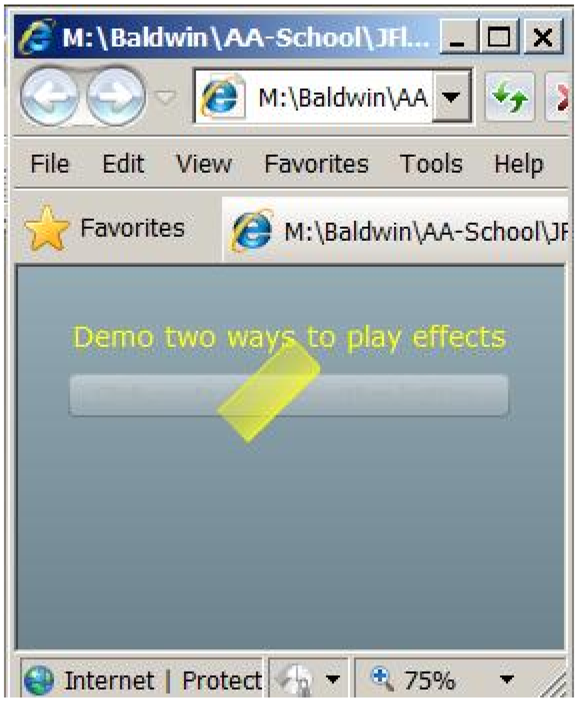

M:\Baldwin\AA-School\JFl...
M:\Baldwin\AA
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JF
Demo two ways to play effects
Internet | Protect
75%

The Glow effect.

You are already familiar with this effect from the program named Effects04 that I explained earlier in this lesson.

Three effects in parallel

Figure 8 shows the three effects being played in parallel.
Three effects in parallel.
Figure


M:\Baldwin\AA-School\JFl...
M:\Baldwin\AA
File Edit View Favorites Tools Help
Favorites
M:\Baldwin\AA-School\JF
Demo two ways to play effects
Internet | Protect
75%

Three effects in parallel.

In this case, the bottom button goes through an interesting gyration before coming to rest in the position shown in Figure 3. Someone once said that a picture is worth a thousand words. In this case, actually running the program is worth a thousand pictures.

Will explain in fragments

As before, I will explain this program in fragments. Aside from the simple MXML file shown in Listing 16, this entire program is written in a class named **Driver** . A complete listing of the **Driver** class is provided in Listing 18.

Two ways to play effects

As I explained earlier , there are at least two different ways to write ActionScript code to play effects:

- Call the **setStyle** method on the target component passing an effect trigger and an effect as parameters to the method as described above . You saw an example of this in the program named Effects04.
- Create an **Effect** object targeted to the component and call the **play** method on the object. This approach doesn't require an effect trigger.

I will illustrate both approaches in this program.

Beginning of the Driver class for Effects05

The driver class for the program named Effects05 begins in Listing 5.

Example:
Beginning of the Driver class for Effects05.

```
package CustomClasses{
  import flash.events.MouseEvent;

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.effects.Glow;
  import mx.effects.Iris;
  import mx.effects.Parallel;
  import mx.effects.Rotate;
  import mx.effects.WipeRight;
  import mx.events.EffectEvent;
  import mx.events.FlexEvent;

  public class Driver extends VBox{
    //Instantiate and save references to most of
the
    // objects needed by the program.
    private var title:Label = new Label();
    private var btnA:Button = new Button();
    private var btnB:Button = new Button();

    private var irisEffect:Iris = new Iris();
    private var wipeEffect:WipeRight = new
WipeRight();
    private var rotateEffect:Rotate = new
Rotate();
    private var glowEffect:Glow = new Glow();

    private var effectCounter:uint = 0;
```

Instantiate four different Effect objects

The most interesting part of Listing 5 is the instantiation of four **different effect objects** :

- **Iris**
- **WipeRight**
- **Rotate**
- **Glow**

The **Iris** effect will be used along with the **setStyle** method to cause the bottom button in Figure 3 to play an **Iris** effect each time it is hidden.

The other three effects in the above list plus an object of the **Parallel** class will be used to apply one of four different effects to the bottom button each time it is shown.

Beginning of the constructor for Effects05

The constructor for the **Driver** class begins in Listing 6.

Example:
Beginning of the constructor for the Effects05.

```
    public function Driver(){//constructor

      //Set title properties and add to the VBox.
      title.setStyle("color","0xFFFF00");
      title.setStyle("fontSize",14);
      title.text = "Demo two ways to play
effects";
      addChild(title);

      //Put labels on the two buttons and disable
one
      // of them.
      btnA.label = "Click me to show the other
button.";
      btnB.label = "Click me to hide me.";
      btnA.enabled = false;//disable btnA at
startup

      //Register click listeners on both buttons,
      // register a show listener on btnB, and add
      // them to the VBox.

btnA.addEventListener(MouseEvent.CLICK,btnAhandler
);

btnB.addEventListener(MouseEvent.CLICK,btnBhandler
);

btnB.addEventListener(FlexEvent.SHOW,showHandler);
      addChild(btnA);
      addChild(btnB);
```

If you have been studying this series of lessons from the beginning, you shouldn't find anything in Listing 6 that you don't understand.

Configure an Iris effect for the bottom button

Listing 7 configures an **Iris** effect that will be played each time the bottom button in Figure 3 is hidden.

Example:
Configure an Iris effect for the bottom button.

```
      irisEffect.duration = 2000;
      irisEffect.addEventListener(

EffectEvent.EFFECT_END,endEffectHandler);

      btnB.setStyle("hideEffect",irisEffect);
```

Note that the bottom button in Figure 3 is referred to by the variable named **btnB** and the top button in Figure 3 is referred to by the variable named **btnA** .

The code in Listing 7 is essentially the same as the code that I explained in the earlier program named Effects04.

Configure three different effects targeted to the bottom button

Listing 8 configures three different **Effect** objects that will be played individually and in combination when the bottom button in Figure 3 is shown.

Example:
Configure three different effects targeted to the bottom button.

```
      //Configure a wipe effect that may be played
      // when btnB is shown.
      wipeEffect.target = btnB;
      wipeEffect.showTarget = true;
      wipeEffect.duration = 2000;

      //Configure a rotate effect that may be
played
      // when btnB is shown.
      rotateEffect.target = btnB;
      rotateEffect.angleFrom = 0;
      rotateEffect.angleTo = 360;
      rotateEffect.duration = 2000;

      //Configure a glow effect that may be played
      // when btnB is shown.
      glowEffect.target = btnB;
      glowEffect.color = 0xFFFF00;
      glowEffect.duration = 4000;
      glowEffect.inner = true;
      glowEffect.strength = 255;

    } //end constructor
```

Different effects require different properties

The three **Effect** objects were instantiated in Listing 5. Different types of effects require that different types of properties be set. However, one property that is common for all types of effects when using this approach is to specify the target component on which the effect is to be played.

> *(Note that it isn't necessary to explicitly specify the target for the earlier approach shown in Listing 7. In that case, the target is the*

> *object on which the **setStyle** method is called.)*

I will leave it as an exercise for the student to go into the documentation and gain an understanding of the behaviors imparted by the different property values in Listing 8.

Listing 8 also signals the end of the constructor.

A click event handler on the bottom button

Let's begin by disposing of the code that is executed when the bottom button is clicked. A click event handler was registered on the bottom button *(**btnB**)* in Listing 6. That event handler is shown in Listing 9.

Example:
A click event handler on the bottom button.

```
    private function
btnBhandler(event:MouseEvent):void{
      btnB.visible = false;
    } //end btnBhandler
```

A hideEffect trigger

The method shown in Listing 9 is executed each time the user clicks the bottom button. The method sets the **visible** property of the bottom button to false. This causes the bottom button to dispatch a **hide** event, which in turn results in a **hideEffect** trigger. As you saw in Listing 7, this causes the program to play an **Iris** effect to hide the button.

An EFFECT_END handler for the Iris effect

When the bottom button is showing, the top button is disabled. Therefore, when the bottom button becomes hidden, the top button must be enabled or there will be no way to show the bottom button again.

Listing 7 registers an event handler on the **Iris** effect that is called each time the effect finishes playing. That event handler is shown in Listing 10.

Example:
An EFFECT_END handler for the Iris effect.

```
    private function endEffectHandler(

event:EffectEvent):void{
      btnA.enabled = true;
    } //end event handler
```

The code in this method sets the **enabled** property of the top button to true making it possible to click that button to show the bottom button again.

That takes care of the code associated with clicking the bottom button.

A click event handler for the top button

Listing 6 registered a **click** event handler on the top button ***(btnA)*** . That event handler is shown in Listing 11. This method is called each time the top button is clicked while it is enabled.

Example:
A click event handler for the top button.

```
    private function
btnAhandler(event:MouseEvent):void{
      btnA.enabled = false;
      btnB.visible = true;
    } //end btnAhandler
```

The code in this method disables the top button and sets the **visible** property of the bottom button to true. This causes the bottom button to dispatch a **show** event.

A **show** event handler was registered on the bottom button in Listing 6.

Beginning of the show event handler registered on the bottom button

That **show** event handler begins in Listing 12.

Example:
Beginning of the Show event handler registered on the bottom button.

```
    private function
showHandler(event:FlexEvent):void{
      //Make certain that none of the effects are
playing.
      wipeEffect.end();
      rotateEffect.end();
      glowEffect.end();

      //Select the effect or effects that will be
      // played.
      if(effectCounter == 0){
        wipeEffect.play();
        effectCounter++;//increment the effect
counter.
```

This method is executed each time the **visible** property of the bottom button is changed from false to true. The transition of that property value from false to true causes the bottom button to dispatch a **show** event.

Stop all effects that may be playing

Listing 12 begins by calling the **end** method on three of the four **Effect** objects that were instantiated in Listing 5. If one of those effects is playing, calling the **end** method on the effect object causes the Flash Player to jump immediately to the end.

Determine which effect to play

Then Listing 12 begins executing a long **if-else-if** statement that determines which effect to play based on the current value of the variable named **effectCounter** that was declared and initialized to a value of zero in Listing 5.

Four possibilities

Depending on the current value of that counter, the program will play one of the following three effects or all three in parallel:

- **WipeRight**
- **Rotate**
- **Glow**

If the current value of the effect counter variable is 0, the last two statements in Listing 12 are executed. Otherwise, control passes to the test at the top of Listing 13.

Play the wipe effect and increment the counter

One of the statements in Listing 12 calls the **play** method on the effect object referred to by **wipeEffect** causing that effect to play. The last statement in Listing 12 increments the effect counter by a value of one. Then control passes to the end of the **showHandler** method near the bottom of Listing 15.

Code to play the Rotate effect

If the value of the effect counter was 1 when control entered the **if-else-if** statement in Listing 12, the last four statements in Listing 13 are executed. Otherwise, control passes to the top of Listing 14.

Example:
Code to play the Rotate effect.

```
      }else if(effectCounter == 1){
        rotateEffect.originX = btnB.width/2;
        rotateEffect.originY = btnB.height/2;
        rotateEffect.play();
        effectCounter++;
```

If the value of the effect counter was 1 when the **if-else-if** statement began execution, the **play** method is called on the **rotateEffect** object by the code in Listing 13.

Establish the center of rotation

Before calling the **play** method, however, the code in Listing 13 establishes the center of the button as the point around which the button will be rotated. It was not possible to establish this point when the **Rotate** effect was configured in Listing 8 because reliable information about the width and height of the button was not yet available.

Could have used creationComplete

Perhaps a more elegant approach to establishing the center of rotation would have been to register a **creationComplete** listener on the **VBox** and to set the values for **originX** in **originY** in that handler. However, that seemed like overkill and I decided to do it in Listing 13.

Increment the counter and go to the end of the method

If the last four statements in Listing 13 are executed, the effect counter is incremented by one. Then control passes to the bottom of the method in Listing 15.

Code to play the Glow effect

If the value of the effect counter was 2 when control entered the **if-else-if** statement in Listing 12, the last two statements in Listing 14 are executed. Otherwise, control passes to the top of Listing 15.

Example:
Code to play the Glow effect.

```
      }else if(effectCounter == 2){
        glowEffect.play();
        effectCounter++;
```

The last two statements in Listing 14 play the glow effect and increment the effect counter. Then control passes to the bottom of the method in Listing 15.

Code to play three effects in parallel

If the value of the effect counter was something other than 0, 1, or 2 when control entered the **if-else-if** statement in Listing 12, the code in the **else** clause in Listing 15 is executed.

Example:
Code to play three effects in parallel.

```
      }else{
        //Play all three effects in parallel.
        var parallel:Parallel = new Parallel();
        parallel.addChild(rotateEffect);
        parallel.addChild(glowEffect);
        parallel.addChild(wipeEffect);
        parallel.play();
        //reset the effect counter
        effectCounter = 0;
      } //end else

    } //end showHandler
    //----------------------------------------------
------//

  } //end class
} //end package
```

An object of the Parallel class

This code instantiates an object of the **Parallel** class and adds the three effects as children of that object. Then the code calls the **play** method on the **Parallel** object. This causes all three effects to play simultaneously.

Reset the counter

Finally, Listing 15 resets the value of the effect counter back to 0 so that the sequence will begin anew the next time the event handler for the **show** event is executed.

Play the same effect on multiple targets simultaneously

You can play the same effect on multiple targets simultaneously by setting the **targets** property on the effect object instead of the **target** object. The **targets** property requires an array containing references to the target objects.

Three steps are required

The following steps are required to play an effect in the Flash Player using this approach.

1. Instantiate and save a reference to an **Effect** object.
2. Set properties on the effect object. Be sure to set the **target** property for a single target or the **targets** property for multiple targets.
3. Call the **play** method on the effect object.

To play multiple effects in parallel or in sequence

1. Instantiate and save references to two or more **Effect** objects.
2. Set properties on the effect objects, being careful to set either the **target** property or the **targets** property.
3. Instantiate a **Parallel** object or a **Sequence** object.
4. Add the effect objects as children of the **Parallel** object or the **Sequence** object.
5. Call the play method on the **Parallel** object or the **Sequence** object.

Note that you can also add **Sequence** objects to **Parallel** objects and vice versa. Just make certain that you don't try to play two instances of the same effect on the same object at the same time.

The end of the program

Listing 15 also signals the end of the **Driver** class and the end of the program.

Run the programs

I encourage you to run these two programs from the web. Then copy the code from Listing 16 through Listing 18. Use that code to create Flex projects. Compile and run the projects. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the MXML and ActionScript files are provided in Listing 16 through Listing 18 below.

Example:
The MXML file used for both programs.

```
<?xml version="1.0" encoding="utf-8"?>

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Example:
The Driver class for Effects04.

```
/*Effects04 11/22/09
This program shows how to set the style on an
object with
 a mouseUpEffect trigger and cause the object to
glow.
*/

package CustomClasses{

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.effects.Glow;

  public class Driver extends VBox{
    //Instantiate and save references to all of
the
    // objects needed by the program.
    private var title:Label = new Label();
    private var button:Button = new Button();
    private var glowEffect:Glow = new Glow();
    //---------------------------------------------
------//

    public function Driver(){//constructor

      //Set title properties and add to the VBox.
      title.setStyle("color","0xFFFF00");
      title.setStyle("fontSize",14);
      title.text = "Demo mouseUpEffect trigger";
      addChild(title);

      button.label = "Click me and watch me
glow.";
      addChild(button);

      glowEffect.color = 0xFF0000;
      glowEffect.strength = 255;
      glowEffect.duration = 10000;
      button.setStyle("mouseUpEffect",glowEffect);

    } //end constructor
    //--------------------------------------------------
------//

  } //end class
} //end package
```

Example:
The Driver class for Effects05.

```
/*Effects05 11/22/09
This program demonstrates two ways to play
effects:
1. Call the play method on the effect.
2. Set the style on an object with a hideEffect
trigger.
**************************************************
*******/
package CustomClasses{
  import flash.events.MouseEvent;

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.effects.Glow;
  import mx.effects.Iris;
  import mx.effects.Parallel;

  import mx.effects.Rotate;
  import mx.effects.WipeRight;
  import mx.events.EffectEvent;
  import mx.events.FlexEvent;

  public class Driver extends VBox{
    //Instantiate and save references to most of
the
    // objects needed by the program.
    private var title:Label = new Label();
    private var btnA:Button = new Button();
    private var btnB:Button = new Button();
    private var irisEffect:Iris = new Iris();
    private var wipeEffect:WipeRight = new
WipeRight();
    private var rotateEffect:Rotate = new
Rotate();
    private var glowEffect:Glow = new Glow();
    private var effectCounter:uint = 0;
    //---------------------------------------------
------//

    public function Driver(){//constructor

      //Set title properties and add to the VBox.
      title.setStyle("color","0xFFFF00");
      title.setStyle("fontSize",14);
      title.text = "Demo two ways to play
effects";
      addChild(title);

      //Put labels on the two buttons and disable
one
      // of them.
      btnA.label = "Click me to show the other
button.";
      btnB.label = "Click me to hide me.";

      btnA.enabled = false;//disable btnA at
startup
```

```
      //Register click listeners on both buttons,
      // register a show listener on btnB, and add
      // them to the VBox.

btnA.addEventListener(MouseEvent.CLICK,btnAhandler
);

btnB.addEventListener(MouseEvent.CLICK,btnBhandler
);

btnB.addEventListener(FlexEvent.SHOW,showHandler);
      addChild(btnA);
      addChild(btnB);

      //Configure an iris effect that will be
played when
      // btnB is hidden.
      irisEffect.duration = 2000;
      irisEffect.addEventListener(

EffectEvent.EFFECT_END,endEffectHandler);
      btnB.setStyle("hideEffect",irisEffect);

      //Configure a wipe effect that may be played
      // when btnB is shown.
      wipeEffect.target = btnB;
      wipeEffect.showTarget = true;
      wipeEffect.duration = 2000;

      //Configure a rotate effect that may be
played
      // when btnB is shown.
      rotateEffect.target = btnB;
      rotateEffect.angleFrom = 0;

      rotateEffect.angleTo = 360;
      rotateEffect.duration = 2000;
      //Configure a glow effect that may be played
      // when btnB is shown.
      glowEffect.target = btnB;
      glowEffect.color = 0xFFFF00;
      glowEffect.duration = 4000;
      glowEffect.inner = true;
      glowEffect.strength = 255;

    } //end constructor
    //--------------------------------------------------
------//

    //This method is executed when btnB is
clicked. It
    // hides itself, which in turn causes the Iris
    // hideEffect to be played on itself.
    private function
btnBhandler(event:MouseEvent):void{
      btnB.visible = false;
    } //end btnBhandler
    //--------------------------------------------------
------//

    //This method is executed when btnA is
clicked. It
    // disables itself and causes btnB to become
visible.
    // This in turn causes btnB to dispatch a show
event
    // which is handled by a different event
handler.
    private function
btnAhandler(event:MouseEvent):void{
      btnA.enabled = false;

      btnB.visible = true;
    } //end btnAhandler
    //--------------------------------------------------
```

```
------//

    //This method is executed when btnB is hidden
and the
    // iris effect ends. It enables btnA so that
the user
    // can click btnA to show btnB again.
    private function endEffectHandler(

event:EffectEvent):void{
      btnA.enabled = true;
    } //end event handler
    //--------------------------------------------
------//

    //This method is executed when btnB becomes
visible
    // and dispatches a show event. It causes any
effects
    // that may be playing to end. Then it one of
three
    // effects or all three in parallel depending
on the
    // value of an effect counter.
    private function
showHandler(event:FlexEvent):void{
      //Make certain that none of the effects are
playing.
      wipeEffect.end();
      rotateEffect.end();
      glowEffect.end();

      //Select the effect or effects that will be
      // played.

      if(effectCounter == 0){
        wipeEffect.play();
        effectCounter++;//increment the effect
counter.
      }else if(effectCounter == 1){
        //Set the rotate origin to the center of
the
        // button. This couldn't be done when the
rotate
        // effect was configured because the true
width
        // and height of the button weren't
available at
        // that time. Another approach would be to
use
        // a creationComplete event handler to set
these
        // values.
        rotateEffect.originX = btnB.width/2;
        rotateEffect.originY = btnB.height/2;
        rotateEffect.play();
        effectCounter++;
      }else if(effectCounter == 2){
        glowEffect.play();
        effectCounter++;
      }else{
        //Play all three effects in parallel.
        var parallel:Parallel = new Parallel();
        parallel.addChild(rotateEffect);
        parallel.addChild(glowEffect);
        parallel.addChild(wipeEffect);
        parallel.play();
        effectCounter = 0;//reset the effect
counter
      } //end else

    } //end showHandler

    //---------------------------------------------
------//
```

```
  } //end class
} //end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Events, Triggers, and Effects
- Files:
    - ActionScript0116\ActionScript0116.htm
    - ActionScript0116\Connexions\ActionScriptXhtml0116.htm

-end-

Creating Custom Effects
Learn how to create a custom effect and how to play it two different ways.

Note: Click CustomEffect02 or CustomEffect03 to run the ActionScript programs from this lesson. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming (OOP) with ActionScript.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the

emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

- Listing 18 . Common MXML code used for both programs.
- Listing 19 . Source code for the class named CustomEffect.
- Listing 20 . Source code for the class named CustomEffectInstance.
- Listing 21 . Driver class for the program named CustomEffect02.
- Listing 22 . Driver class for the program named CustomEffect03.

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

In an earlier lesson titled **Events, Triggers, and Effects** , I taught you how to use the triggers and effects that are built into the ActionScript language.

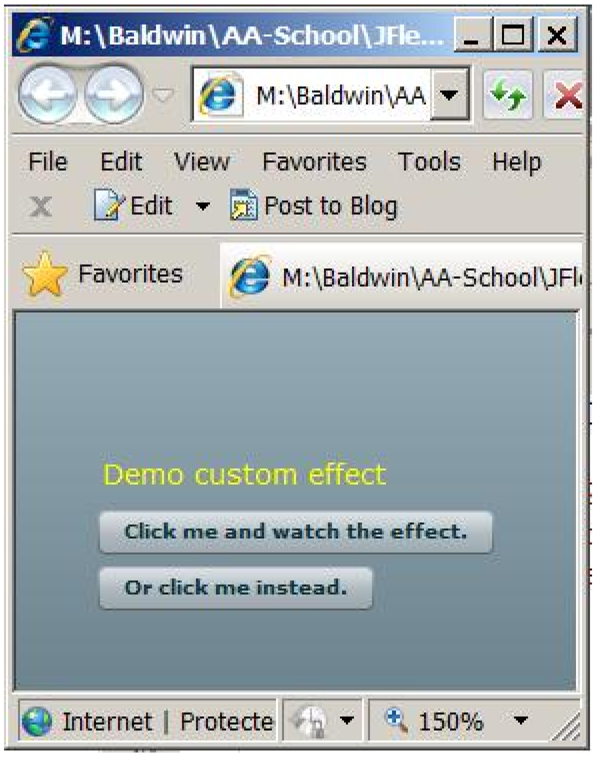
In this lesson, I will teach you how to create your own custom effects. I will also explain two different programs that use a custom effect of my own design. I recommend that you run the online version of each of the two programs before continuing with the lesson.

Preview

Program output at startup

Figure 1 shows the screen output of both programs at startup.
Program output at startup.

Figure



Demo custom effect

Click me and watch the effect.

Or click me instead.

Program output at startup.

A common custom effect

Both programs apply the same custom effect to both buttons. However, the program named **CustomEffect02** applies the effect in such a way that it is played on both buttons simultaneously if either button is clicked. The program named **CustomEffect03** applies the custom effect in such a way that it plays individually on each button when the button is clicked. I will explain the reason for this difference later.

One run is worth a thousand pictures

Hopefully by now you have been able to run the online version of both programs because the effect is difficult to explain. Basically, the custom effect consists of the parallel execution of three types of standard ActionScript effects:

- WipeRight
- Glow
- Rotate

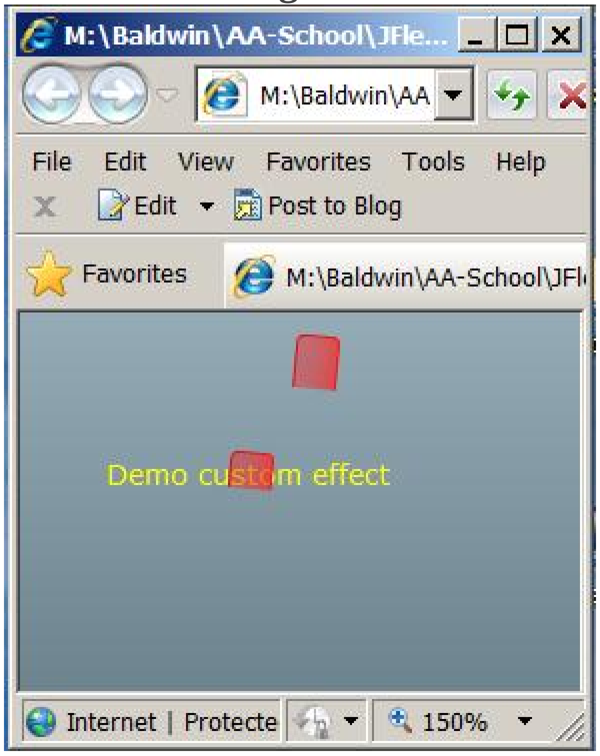
The effect appears to cause the buttons to leave their positions, change color, and fly around for a short while before settling back into their normal positions.

CustomEffect02 output after clicking a button

Figure 2 shows the output from the program named CustomEffect02 shortly after clicking one of the buttons.

CustomEffect02 output after clicking a button.

Figure



CustomEffect02 output after clicking a button.

Discussion and sample code

Will discuss in fragments

I will discuss and explain these two programs in fragments. Complete listings of the MXML code and the ActionScript code are provided near the end of the lesson beginning with Listing 18.

The MXML files

Both programs use the same simple MXML code shown in Listing 1.

Example:
Common MXML Code.

```
<?xml version="1.0" encoding="utf-8"?>

<!--CustomEffect02 11/26/09
Illustrates a custom effect, which is the parallel
playing of three standard effects:

WipeRight
Rotate
Glow

The effect is applied to two buttons each time
either
button is clicked.

This version sets the targets on the effect and
calls the
play method on the effect.

See the Flex 3 Cookbook, page 363
Also see
http://livedocs.adobe.com/flex/3/html/help.html?
```

```
content=createeffects_2.html#178126
Also see
http://livedocs.adobe.com/flex/3/html/help.html?
content=behaviors_04.html#275399
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

An object of the class named Driver

As you can see, this MXML file simply instantiates an object of the class named **Driver** . That's because almost all of the code in these two programs is written in ActionScript instead of MXML.

Creating a custom effect

You must define two classes to create a custom effect. One class is a *factory* class that extends the class named **Effect** . The other class is an *instance* class that extends the class named **EffectInstance** .

The instance class plays the effect

When the time comes to play the effect on a component, the factory class instantiates an object of the instance class to actually play the effect. If the same effect is played on two or more components at the same time, a different object of the instance class is instantiated to play the effect on each component.

Play three effects in parallel

As explained in Listing 1, the custom class that I designed for use in this lesson plays the following three effects in parallel:

- WipeRight
- Rotate
- Glow

You learned about something similar to this in my earlier lesson titled **Events, Triggers, and Effects** . However, in that lesson I didn't combine the three effects into a single custom effect the way that I will in this lesson.

Knowledge of OOP is required

I will do the best that I can to explain this code. Even at that, you are likely to need a pretty good understanding of object-oriented programming to understand the code required to create a custom effect. As you will see later, the required code is steeped in overridden methods, interfaces, and other object-oriented concepts.

The class named CustomEffect

The class named **CustomEffect** begins in Listing 2. A complete listing of the class is provided in Listing 19 near the end of the lesson.

Example:
Beginning of the class named CustomEffect.

```
package CustomClasses{
  import mx.effects.Effect;
  import mx.effects.IEffectInstance;
  import mx.events.EffectEvent;

  public class CustomEffect extends Effect{
    //Would prefer to make these private and use
implicit
    // setter methods, but I decided to leave them
public
    // to simplify the code.
    public var theDuration:Number = 2000;//default
value
    public var rotateAngleFrom:Number =
0;//default value
    public var rotateAngleTo:Number =
360;//default value
    public var wipeShowTarget:Boolean =
true;//default
    public var glowColor:uint = 0xFF0000;//default
value
    public var glowInner:Boolean = true;//default
value
    public var glowStrength:Number = 255;//default
value
```

The factory class

Of the two required classes, this is the factory class that I mentioned earlier. This class must extend the class named **Effect** , and will override methods inherited from that class.

Public instance variables

Listing 2 declares and initializes seven public instance variables that will be used to set properties on the **WipeRight** object, the **Rotate** object, and the **Glow** object. I provided default values for these variables so that the program will work even if the driver program fails to provide the required values.

Could use implicit setter methods

As I mentioned in the comments, I would prefer to make these variables private and provide an implicit setter method for each variable. However, I decided to make them public to simplify the code and make it easier to explain.

The constructor for the class named CustomEffect

The constructor is shown in its entirety in Listing 3.

Example:
The constructor for the class named CustomEffect.

```
    public function 
CustomEffect(target:Object=null){
      super(target);
      instanceClass = CustomEffectInstance;
    } //end constructor
```

The incoming parameter

The incoming parameter for the constructor is the generic type **Object** . This parameter must specify the component on which the effect is to be played.

If no target is passed as a parameter to the constructor, the default null value prevails and the **target** property of the object must be set. As you will see later, an alternative property named **targets** can be set to cause the effect to be played on multiple targets at the same time.

Call the superclass constructor

Without attempting to explain why, I am going to tell you that it is frequently necessary in OOP to cause the constructor for a class to make a call to the constructor of its superclass as the first statement in the constructor. This constructor is no exception to that rule.

The first statement in Listing 3 is a call to the constructor for the **Effect** class passing a reference to the target component(s) as a parameter. When that constructor returns control, the second statement in Listing 3 is executed.

The instanceClass property

This class inherits a property named **instanceClass** from the class named **Effect** . According to About creating a custom effect , the factory class that you define must set the value of this property to the name of the instance class that will be used to play the effect.

In this program, the name of the instance class is **CustomEffectInstance** , as shown in Listing 3. I will explain the code in that class after I finish explaining the code in this class.

This inherited property provides the mechanism that ties the instance class to the factory class.

Override the initInstance method

Listing 4 overrides an inherited method named **initInstance** .

Example:
Override the initInstance method.

```
    override protected function initInstance(

instance:IEffectInstance):void{
      super.initInstance(instance);
      CustomEffectInstance(instance).theDuration =

this.theDuration;

CustomEffectInstance(instance).rotateAngleFrom =

this.rotateAngleFrom;
      CustomEffectInstance(instance).rotateAngleTo
=

this.rotateAngleTo;

CustomEffectInstance(instance).wipeShowTarget =

this.wipeShowTarget;
      CustomEffectInstance(instance).glowColor =

this.glowColor;
      CustomEffectInstance(instance).glowInner =

this.glowInner;
      CustomEffectInstance(instance).glowStrength
=

this.glowStrength;

    } //end initInstance
```

Set the property values in the instance object

According to About creating a custom effect , the purpose of this method is to copy property values from the factory class to the instance class. Flex calls this method from the **Effect.createInstance()** method. You don't have to call it yourself, but you must prepare it to be called.

A reference to the instance object as type iEffectInstance

The **initInstance** method receives a reference to the instance object as the interface type **IEffectInstance** . The objective is to write values into the properties belonging to the instance object. However, the **IEffectInstance** interface doesn't know anything about properties having those names. Therefore, it is necessary to cast the instance object's reference to the type of the instance object before making each assignment. One such cast operation is shown by the statement that begins with **CustomEffectInstance(instance)** in Listing 4.

Call the initInstance method of the superclass

Also note that you must call the **initInstance** method of the superclass in your overridden method as shown in Listing 4.

This method provides the mechanism by which required property values make it all the way from the driver class to the instance class.

Override the getAffectedProperties method

According to About creating a custom effect , you must override the inherited method named **getAffectedProperties** in such a way as to return an array of strings. Each string is the name of a property of the target object that is changed by the effect. If no properties are changed, you must return an empty array.

Listing 5 shows my overridden version of the **getAffectedProperties** method.

Example:
Override the getAffectedProperties method.

```
    override public function

getAffectedProperties():Array{
      return
["rotation","rotationX","rotationY","x","y"];
    } //end getAffectedProperties
    //----------------------------------------
------//
  } //end class
} //end package
```

This is a little difficult

It is a little difficult to know exactly which properties belonging to the target component will be modified by the effect, particularly when the custom effect is a composite of existing effects. Also, I don't know whether a change must be permanent or whether a temporary change in the value of a property requires that it be returned by the **getAffectedProperties** method. There are several target property values that are temporarily changed by this custom effect.

In this program, the target component is a **Button** object but it could be any component. I went through the list of properties belonging to a button and came up with the five shown in Listing 5 as those most likely to be modified.

The end of the CustomEffect class

Listing 5 also signals the end of the class named **CustomEffect** . In addition to the methods that were overridden above, the following two inherited methods may optionally be overridden as well:

- **effectStartHandler** - called when the effect instance starts playing.
- **effectEndHandler** - called when the effect instance finishes playing.

As the names and descriptions of these two methods suggest, they can be overridden to provide any special behavior that you need when the effect starts and finishes playing.

The class named CustomEffectInstance

The class named **CustomEffectInstance** begins in Listing 6. A complete listing of the class is provided in Listing 20 near the end of the lesson.

Example:
Beginning of the CustomEffectInstance class.

```
package CustomClasses{
  import mx.effects.EffectInstance;
  import mx.effects.Glow;
  import mx.effects.Parallel;
  import mx.effects.Rotate;
  import mx.effects.WipeRight;
  import mx.events.FlexEvent;

  public class CustomEffectInstance
                                   extends
EffectInstance{
    //Instantiate the individual effects that will
be
    // combined in parallel to produce the custom
effect.
    private var wipeEffect:WipeRight = new
WipeRight();
    private var rotateEffect:Rotate = new
Rotate();
    private var glowEffect:Glow = new Glow();
```

Extends the class named EffectInstance

This class extends the class named **EffectInstance** . As before, the code in this class will override methods inherited from the **EffectInstance** class.

Instantiate and save references to standard effects

Listing 6 instantiates and saves references to the **WipeRight** , **Rotate** , and **Glow** effect classes. They will be combined to run concurrently later in the program.

Declare variables for storage of effect properties

Listing 7 declares a set of variables that will be used to store the properties for the three different effects that are combined to create a composite effect.

Example:
Declare variables for storage of effect properties.

```
    //Variables for the storage of effect
properties.
    public var theDuration:Number;
    public var rotateAngleFrom:Number;
    public var rotateAngleTo:Number;
    public var wipeShowTarget:Boolean;
    public var glowColor:uint;
    public var glowInner:Boolean;
    public var glowStrength:Number;
```

Property values are stored in these variables by the code in the **initInstance** method shown in Listing 4.

The constructor for the CustomEffectInstance class

The constructor for the class is shown in its entirety in Listing 8.

Example:

The constructor for the CustomEffectInstance class.

```
    public function CustomEffectInstance(

theTarget:Object){
      super(theTarget);

      //Set the target for all three individual
effects.
      rotateEffect.target = theTarget;
      wipeEffect.target = theTarget;
      glowEffect.target = theTarget;
    } //end constructor
```

The target component

As was the case in Listing 3, this constructor receives a parameter of the generic type **Object** , which specifies the component on which the effect will be played.

A different EffectInstance object for each target component

If multiple target components are specified by setting the **targets** property of the **CustomEffect** class, different objects of the **CustomEffectInstance** class are instantiated and targeted to the different components in the list of target components.

Target the WipeRight, Rotate, and Glow effects to the target component

The code in the constructor sets the specified target to be the target for the three types of standard effects that will be played concurrently.

Override the inherited play method

You may have noticed that the code in the **CustomEffect** class didn't include any of the operational details regarding the nature of the custom effect. Those details are programmed into an overridden **play** method that begins in Listing 9.

Example:
Override the inherited play method.

```
    override public function play():void{
      super.play();

      //Note: The following values cannot be set
in the
      // constructor because the variables aren't
stable
      // at that point in time.

      //Configure the rotate effect
      rotateEffect.angleFrom = rotateAngleFrom;
      rotateEffect.angleTo = rotateAngleTo;
      rotateEffect.duration = theDuration;

      //Configure the wipe effect.
      wipeEffect.showTarget = wipeShowTarget;
      wipeEffect.duration = theDuration;

      //Configure the glow effect.
      glowEffect.color = glowColor;
      glowEffect.duration = theDuration;
      glowEffect.inner = glowInner;
      glowEffect.strength = glowStrength;
```

The overridden play method produces the desired effect

Later on you will see that the driver class for this program instantiates an object of the custom effect class and calls the **play** method on that object. At that point, the driver class will be calling the method that begins in Listing 9.

Set the required properties on the three standard effects

Listing 9 uses the values that were stored in the variables in Listing 7 by the initInstance method in Listing 4 to set the required properties for each of the three individual effects that will be combined to produce this custom effect.

Ordinarily, you might think that this could have been accomplished in the constructor for the class. However, the values in the variables in Listing 7 aren't stable until the constructor has finished constructing the object. Therefore, it is necessary to defer the assignments in Listing 9 until after the construction of the object is complete.

Play the three effects in parallel

You learned how to use an object of the **Parallel** class to play two or more effects in parallel in the earlier lesson titled **Events, Triggers, and Effects** .

Example:
Play the three effects in parallel.

```
      //Play all three effects in parallel.
      var parallel:Parallel = new Parallel();
      parallel.addChild(rotateEffect);
      parallel.addChild(glowEffect);
      parallel.addChild(wipeEffect);
      parallel.play();
    } //end play
    //----------------------------------------------
------//
  } //end class
} //end package
```

Therefore, you shouldn't have any difficulty understanding the code in Listing 10.

Steps required to create a custom effect

You must define a factory class and an instance class. The following steps are required to create and prepare the factory class:

- Define a factory class that extends the **Effect** class.
- Declare variables in the factory class, if any are required, to store the property values for the custom effect.
- Define a constructor for the factory class that calls the constructor for the superclass and also sets the name of the instance class into the inherited variable named **instanceClass** .
- Override the **initInstance** method in the factory class to store the property values into variables in the instance class. Also call the **initInstance** method of the superclass in that overridden method.
- Override the **getAffectedProperties** method in the factory class to return a list of target component properties that will be modified by the effect. Return an empty array if none will be modified.

Define and prepare the instance class

Having defined the factory class using the steps listed above, define the instance class by performing the following steps:

- Define an instance class that extends the **EffectInstance** class.
- Declare public instance variables for the storage of effect property values if any are required.
- Define a constructor for the instance class that deals appropriately with the target component.
- Override the inherited **play** method to implement the actual behavior of the custom effect.

The end of the CustomEffectInstance class

Listing 10 signals the end of the class named **CustomEffectInstance** .

The Driver class for the program named CustomEffect02

The two classes discussed above constitute the whole of the custom effect. I will provide and explain two different driver classes that use the same custom effect but use it in different ways. The driver class for the program named **CustomEffect02** begins in Listing 11. A complete listing of this class is provided in Listing 21 near the end of the lesson.

Two ways to play effects

You learned in the earlier lesson titled **Events, Triggers, and Effects** that there are at least two different ways to cause an effect to be played on a component in an ActionScript program. One way is to call the **setStyle** method on the component and associate an effect trigger with an effect. With that approach, the effect will be played each time the effect trigger fires.

The second way

The second way to play an effect on a component is to target an **Effect** object to the component and then call the **play** method on the effect object. This approach doesn't make explicit use of the effect trigger.

I will illustrate the second approach in the program named **CustomEffect02** , and will illustrate the first approach later in the program named **CustomEffect03** .

Beginning of the Driver class for CustomEffect02

The Driver class begins in Listing 11.

Example:
Beginning of the Driver class for CustomEffect02.

```
package CustomClasses{

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.controls.Spacer;
  import flash.events.MouseEvent;

  public class Driver extends VBox{
    //Instantiate and save references to all of
the
    // objects needed by the program.
    private var title:Label = new Label();
    private var btnA:Button = new Button();
    private var btnB:Button = new Button();
    private var spacer:Spacer = new Spacer();
    private var theEffect:CustomEffect =
                                       new
CustomEffect();
```

The code in Listing 11 extends the **VBox** class and instantiates objects for all of the components that will be required to produce the GUI shown in Figure 1. In addition, Listing 11 instantiates an object of the new **CustomEffect** class.

No target is passed to the constructor

As you can see from Listing 11, a target component was not passed to the constructor for the **CustomEffect** class. Instead, an alternative approach that sets the **targets** property will be used.

Beginning of the constructor for the Driver class

The constructor for the Driver class begins in Listing 12.

Example:
Beginning of the constructor for the Driver class.

```
    public function Driver(){//constructor
      //Make some space at the top of the display.
      spacer.height = 40;
      addChild(spacer);

      //Set title properties and add to the VBox.
      title.setStyle("color","0xFFFF00");
      title.setStyle("fontSize",14);
      title.text = "Demo custom effect";
      addChild(title);

      //Instantiate two buttons and add them to
the VBox.
      // Register the same event listener on both
of
      // them.
      btnA.label = "Click me and watch the
effect.";

btnA.addEventListener(MouseEvent.CLICK,handler);
      addChild(btnA);

      btnB.label = "Or click me instead.";

btnB.addEventListener(MouseEvent.CLICK,handler);
      addChild(btnB);
```

There is nothing new in Listing 12 so further explanation shouldn't be required. It is worth noting, however, that the same **click** event listener is registered on both buttons.

Set properties on the custom effect

Listing 13 shows the code that sets properties on the custom effect.

Example:
Set properties on the custom effect.

```
      //Specify both buttons to be the target for
the
      // same effect.
      theEffect.targets = [btnA,btnB];

      //Set various properties needed by the
effect.
      theEffect.theDuration = 4000;
      theEffect.rotateAngleFrom = 0;
      theEffect.rotateAngleTo = 720;
      theEffect.wipeShowTarget = true;
      theEffect.glowColor = 0xFF0000;
      theEffect.glowInner = true;
      theEffect.glowStrength = 255;

    } //end constructor
```

With the exception of the property named **targets** , the values that are assigned in Listing 13 are stored in the variables that are declared in Listing 2.

The targets property

The **targets** property is inherited into the **CustomEffect** class from the **Effect** class. Note that both buttons are passed to the **targets** property in the form of an array containing references to the two buttons. This causes the custom effect to be played on both buttons at the same time.

Listing 13 also signals the end of the constructor for the **Driver** class.

The common click event handler

The click event handler that is registered on both buttons is shown in Listing 14.

Example:
The common click event handler.

```
    public function 
handler(event:MouseEvent):void{
      theEffect.play();
    }//end handler

  } //end class
} //end package
```

The event handler calls the **play** method on the custom effect object whenever either of the buttons shown in Figure 1 is clicked. This causes the **play** method defined in Listing 9 to be executed.

The end of the program

Listing 14 also signals the end of the program named **CustomEffect02** .

The Driver class for the program named CustomEffect03

The **Driver** class for the program named **CustomEffect03** begins in Listing 15. A complete listing of the class is provided in Listing 22 near the end of the lesson. This program uses the first approach for playing an effect.

Example:
Beginning of the Driver class for CustomEffect03.

```
package CustomClasses{

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.controls.Spacer;

  public class Driver extends VBox{
    //Instantiate and save references to all of
the
    // objects needed by the program.
    private var title:Label = new Label();
    private var btnA:Button = new Button();
    private var btnB:Button = new Button();
    private var spacer:Spacer = new Spacer();
    private var theEffect:CustomEffect =
                                       new
CustomEffect();
    //-----------------------------------------------
------//

    public function Driver(){//constructor
      //Make some space at the top of the display.
      spacer.height = 40;
      addChild(spacer);

      //Set title properties and add to the VBox.
      title.setStyle("color","0xFFFF00");
      title.setStyle("fontSize",14);
      title.text = "Demo custom effect";
      addChild(title);

      //Instantiate two buttons and add them to
the VBox.
```

```
      // Register the same event listener on both
of
      // them.
      btnA.label = "Click me and watch the
effect.";
      addChild(btnA);

      btnB.label = "Or click me instead.";
      addChild(btnB);
```

Very similar to the previous code

The code in Listing 15 matches the code in Listing 11 and Listing 12 with a few exceptions:

- There is no import directive for the **MouseEvent** class.
- There are no **click** event handlers registered on the buttons.

Set properties on the custom effect

Listing 16 sets the properties on the custom effect.

Example:
Set properties on the custom effect.

```
      //Set various properties needed by the
effect.
      theEffect.theDuration = 4000;
      theEffect.rotateAngleFrom = 0;
      theEffect.rotateAngleTo = 720;
      theEffect.wipeShowTarget = true;
      theEffect.glowColor = 0xFF0000;
      theEffect.glowInner = true;
      theEffect.glowStrength = 255;
```

Once again, this code is very similar to the code in Listing 13. There is one major difference, however. The **targets** property for the effect is not explicitly set to the buttons as is the case in Listing 13.

Apply the effect to the two buttons individually

Listing 17 shows the major difference between the two programs.

Example:
Apply the effect to the two buttons individually.

```
      btnA.setStyle("mouseUpEffect",theEffect);
      btnB.setStyle("mouseUpEffect",theEffect);

    } //end constructor
    //--------------------------------------------------
------//
  } //end class
} //end package
```

Use the setStyle method and the effect trigger

Whereas the previous program explicitly sets the buttons as targets of the effect and calls the **play** method on the effect, this program uses the **setStyle** approach and associates the custom effect with a **mouseUpEffect** trigger on each button individually. As a result, when the mouse button is released while the mouse pointer is over one of the buttons, the effect is played on that button alone.

May not be possible to specify multiple targets

I don't know of any easy way to use this approach to cause the effect to be played on two or more components at the same time. The documentation hints that this may not be possible.

The end of the program

Listing 17 also signals the end of the **Driver** class and the end of the program.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 18 through Listing 22. Use that code to create Flex projects. Compile and run the projects. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the Flex applications discussed in this lesson are provided below.

Example:
Common MXML code used for both programs.

```
<?xml version="1.0" encoding="utf-8"?>
```

```
<!--CustomEffect02 11/26/09
Illustrates a custom effect, which is the parallel
playing of three standard effects:

WipeRight
Rotate
Glow

The effect is applied to two buttons each time
either
button is clicked.

This version sets the targets on the effect and
calls the
play method on the effect.

See the Flex 3 Cookbook, page 363
Also see
http://livedocs.adobe.com/flex/3/html/help.html?
content=createeffects_2.html#178126
Also see
http://livedocs.adobe.com/flex/3/html/help.html?
content=behaviors_04.html#275399
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Example:
Source code for the class named CustomEffect.

```
package CustomClasses{
  import mx.effects.Effect;
  import mx.effects.IEffectInstance;
  import mx.events.EffectEvent;

  public class CustomEffect extends Effect{

    //Would prefer to make these private and use
implicit
    // setter methods, but I decided to leave them
public
    // to simplify the code.
    public var theDuration:Number = 2000;//default
value
    public var rotateAngleFrom:Number =
0;//default value
    public var rotateAngleTo:Number =
360;//default value
    public var wipeShowTarget:Boolean =
true;//default
    public var glowColor:uint = 0xFF0000;//default
value
    public var glowInner:Boolean = true;//default
value
    public var glowStrength:Number = 255;//default
value

    public function
CustomEffect(target:Object=null){
      super(target);
      instanceClass = CustomEffectInstance;
    } //end constructor

    override protected function initInstance(
instance:IEffectInstance):void{
      super.initInstance(instance);

      CustomEffectInstance(instance).theDuration =

this.theDuration;

CustomEffectInstance(instance).rotateAngleFrom =

this.rotateAngleFrom;
      CustomEffectInstance(instance).rotateAngleTo
=

this.rotateAngleTo;

CustomEffectInstance(instance).wipeShowTarget =

this.wipeShowTarget;
      CustomEffectInstance(instance).glowColor =

this.glowColor;
      CustomEffectInstance(instance).glowInner =

this.glowInner;
      CustomEffectInstance(instance).glowStrength
=

this.glowStrength;

    } //end initInstance
    //---------------------------------------------
------//

    override public function

getAffectedProperties():Array{
      return
["rotation","rotationX","rotationY","x","y"];
    } //end getAffectedProperties
    //----------------------------------------------------
------//
  } //end class
} //end package
```

Example:
Source code for the class named CustomEffectInstance.

```
package CustomClasses{
  import mx.effects.EffectInstance;
  import mx.effects.Glow;
  import mx.effects.Parallel;
  import mx.effects.Rotate;
  import mx.effects.WipeRight;
  import mx.events.FlexEvent;

  public class CustomEffectInstance
                                   extends
EffectInstance{
    //Instantiate the individual effects that will
be
    // combined in parallel to produce the custom
effect.
    private var wipeEffect:WipeRight = new
WipeRight();
    private var rotateEffect:Rotate = new
Rotate();
    private var glowEffect:Glow = new Glow();

    //Variables for the storage of effect
properties.
```

```
    public var theDuration:Number;
    public var rotateAngleFrom:Number;
    public var rotateAngleTo:Number;
    public var wipeShowTarget:Boolean;
    public var glowColor:uint;
    public var glowInner:Boolean;
    public var glowStrength:Number;

    public function CustomEffectInstance(

theTarget:Object){
      super(theTarget);

      //Set the target for all three individual
effects.
      rotateEffect.target = theTarget;
      wipeEffect.target = theTarget;
      glowEffect.target = theTarget;
    } //end constructor

    override public function play():void{
      super.play();

      //Note: The following values cannot be set
in the
      // constructor because the variables aren't
stable
      // at that point in time.

      //Configure the rotate effect
      rotateEffect.angleFrom = rotateAngleFrom;
      rotateEffect.angleTo = rotateAngleTo;
      rotateEffect.duration = theDuration;

      //Configure the wipe effect.
      wipeEffect.showTarget = wipeShowTarget;
      wipeEffect.duration = theDuration;
      //Configure the glow effect.
      glowEffect.color = glowColor;
      glowEffect.duration = theDuration;
      glowEffect.inner = glowInner;
      glowEffect.strength = glowStrength;

      //Play all three effects in parallel.
      var parallel:Parallel = new Parallel();
      parallel.addChild(rotateEffect);
      parallel.addChild(glowEffect);
      parallel.addChild(wipeEffect);
      parallel.play();
    } //end play
    //--------------------------------------------------
------//
  } //end class
} //end package
```

Example:
Driver class for the program named CustomEffect02.

```
/*CustomEffect02 11/26/09
**************************************************
*******/

package CustomClasses{

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.controls.Spacer;
  import flash.events.MouseEvent;
  public class Driver extends VBox{
    //Instantiate and save references to all of
the
    // objects needed by the program.
    private var title:Label = new Label();
    private var btnA:Button = new Button();
    private var btnB:Button = new Button();
    private var spacer:Spacer = new Spacer();
    private var theEffect:CustomEffect =
                                       new
CustomEffect();
    //---------------------------------------------
------//

    public function Driver(){//constructor
      //Make some space at the top of the display.
      spacer.height = 40;
      addChild(spacer);

      //Set title properties and add to the VBox.
      title.setStyle("color","0xFFFF00");
      title.setStyle("fontSize",14);
      title.text = "Demo custom effect";
      addChild(title);

      //Instantiate two buttons and add them to
the VBox.
      // Register the same event listener on both
of
      // them.
      btnA.label = "Click me and watch the
effect.";

btnA.addEventListener(MouseEvent.CLICK,handler);
      addChild(btnA);

      btnB.label = "Or click me instead.";
btnB.addEventListener(MouseEvent.CLICK,handler);
      addChild(btnB);

      //Specify both buttons to be the target for
the
      // same effect.
      theEffect.targets = [btnA,btnB];

      //Set various properties needed by the
effect.
      theEffect.theDuration = 4000;
      theEffect.rotateAngleFrom = 0;
      theEffect.rotateAngleTo = 720;
      theEffect.wipeShowTarget = true;
      theEffect.glowColor = 0xFF0000;
      theEffect.glowInner = true;
      theEffect.glowStrength = 255;

    } //end constructor
    //-----------------------------------------------
------//

    public function
handler(event:MouseEvent):void{
      theEffect.play();
    }//end handler

  } //end class
} //end package
```

Example:
Driver class for the program named CustomEffect03.

```
/*CustomEffect03 11/27/09
**************************************************
*******/

package CustomClasses{

  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Label;
  import mx.controls.Spacer;

  public class Driver extends VBox{
    //Instantiate and save references to all of
the
    // objects needed by the program.
    private var title:Label = new Label();
    private var btnA:Button = new Button();
    private var btnB:Button = new Button();
    private var spacer:Spacer = new Spacer();
    private var theEffect:CustomEffect =
                                       new
CustomEffect();
    //----------------------------------------------
------//

    public function Driver(){//constructor
      //Make some space at the top of the display.
      spacer.height = 40;
      addChild(spacer);

      //Set title properties and add to the VBox.
      title.setStyle("color","0xFFFF00");
      title.setStyle("fontSize",14);
      title.text = "Demo custom effect";
      addChild(title);

      //Instantiate two buttons and add them to
the VBox.
      // Register the same event listener on both
of
      // them.
      btnA.label = "Click me and watch the
effect.";
      addChild(btnA);

      btnB.label = "Or click me instead.";
      addChild(btnB);

      //Set various properties needed by the
effect.
      theEffect.theDuration = 4000;
      theEffect.rotateAngleFrom = 0;
      theEffect.rotateAngleTo = 720;
      theEffect.wipeShowTarget = true;
      theEffect.glowColor = 0xFF0000;
      theEffect.glowInner = true;
      theEffect.glowStrength = 255;

      //Apply the effect to the two buttons
individually.
      btnA.setStyle("mouseUpEffect",theEffect);
      btnB.setStyle("mouseUpEffect",theEffect);

    } //end constructor
    //--------------------------------------------
------//
  } //end class
} //end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Creating Custom Effects
- Files:
    - ActionScript0118\ActionScript0118.htm
    - ActionScript0118\Connexions\ActionScriptXhtml0118.htm

-end-

Bitmap Basics
Learn about the different requirements for displaying bitmap data in Flex projects and ActionScript projects.

Note: Click Bitmap03 or Bitmap04 to run the ActionScript programs from this lesson. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming (OOP) with ActionScript. The purpose of this particular lesson is to provide you with about fifteen minutes of instruction that may save you hours of debugging when you first start creating programs that use the **Bitmap** and **BitmapData** classes.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Most of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the

emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

The ActionScript documentation provides sample programs on the following web pages that cannot be run without modification in a Flex application, even when the entire application is programmed in **ActionScript** .

- Manipulating pixels
- Bitmap

The problem is that these two sample programs were written to be compiled and run as pure ActionScript programs. There are some additional requirements that you must adhere to when writing bitmap programs to run as Flex applications.

Preview

I will explain two sample programs in this lesson. Both programs are scaled down versions of the sample program provided in the first link listed above .

The first program that I will explain can be compiled and run as a pure ActionScript program. The second program can be compiled and run as a Flex application with the majority of the code being written in ActionScript.

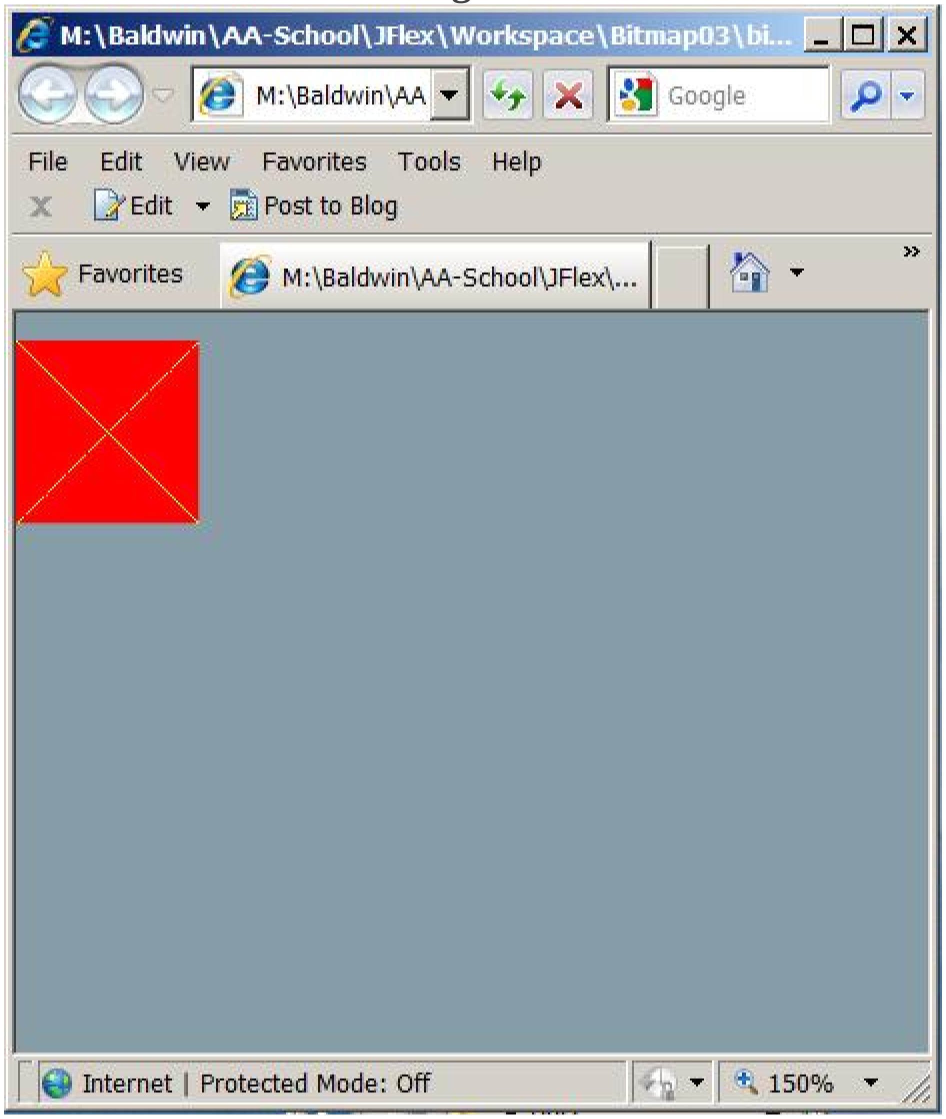
Run the online version of the programs

I encourage you to run the online versions of the two programs before continuing with this discussion.

Screen output from the program named Bitmap03

Both programs are designed to display a red square, 100 pixels on a side, with a yellow cross in the square. The screen output for the pure ActionScript program named **Bitmap03** is shown in Figure 1. *(Note that the yellow cross is much more visible in the original than in this screen shot.)* Screen output from the program named Bitmap03.

Figure


M:\Baldwin\AA-School\JFlex\Workspace\Bitmap03\bi...
M:\Baldwin\AA
Google
File
Edit
View
Favorites
Tools
Help
Edit
Post to Blog
Favorites
M:\Baldwin\AA-School\JFlex\...
Internet | Protected Mode: Off
150%

Screen output from the program named Bitmap03.

The size and position of the square varies

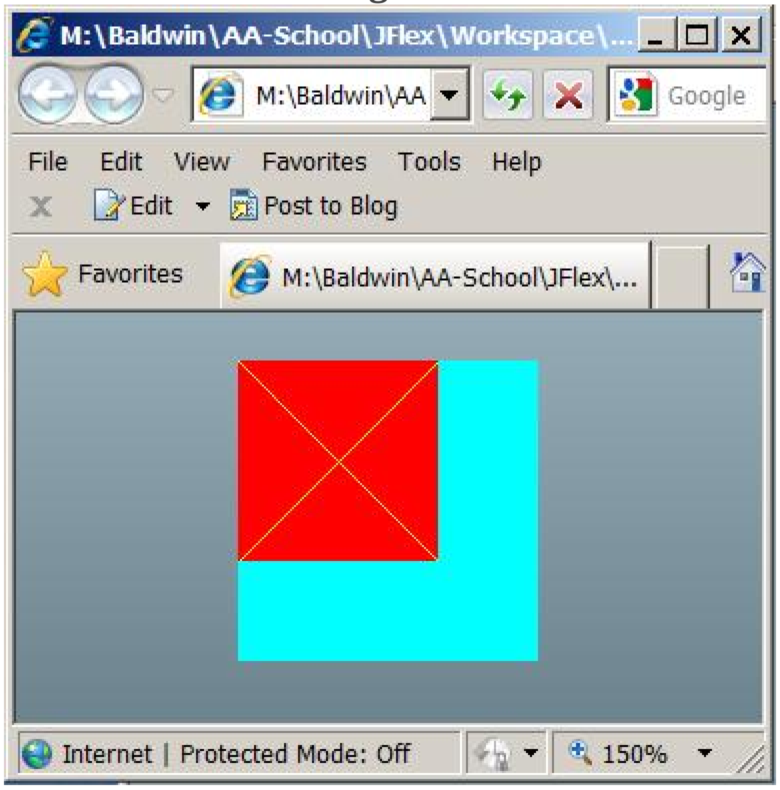
If you run the online version of this program, you will note that the size and position of the square varies depending on the width and height of the browser window.

Screen output from the program named Bitmap04

The program named **Bitmap04** places the red square in a **VBox** container with a cyan background. The screen output is shown in Figure 2.

Screen output from the program named Bitmap04.

Figure


M:\Baldwin\AA-School\JFlex\Workspace\...
M:\Baldwin\AA
Google
File
Edit
View
Favorites
Tools
Help
Edit
Post to Blog
Favorites
M:\Baldwin\AA-School\JFlex\...
Internet | Protected Mode: Off
150%

Screen output from the program named Bitmap04.

Although the **VBox** moves horizontally to remain centered in the browser window, neither the **VBox** nor the red square change size as the width and height of the browser window is changed.

Discussion and sample code

I will explain two programs in the remainder of this lesson.

The program named Bitmap03

Will discuss in fragments

I will explain the code for these two programs in fragments. Complete listings of all the code for both of the programs are provided beginning with Listing 8 near the end of the lesson.

Beginning of the class named Bitmap03

This program consists of a single class definition file named **Bitmap03** , which begins in Listing 1.

Example:
Beginning of the class named Bitmap03.

```
package {
  import flash.display.Sprite
  import flash.display.Bitmap;
  import flash.display.BitmapData;

  public class Bitmap03 extends Sprite {

    public function Bitmap03(){
      var bitmapData:BitmapData =
                new BitmapData(100, 100, false,
0xFF0000);
```

The two main bitmap classes

The two main ActionScript classes that are used with bitmaps are:

- BitmapData
- Bitmap

What is the difference between the two classes?

Briefly, an object of the **BitmapData** class encapsulates the actual pixel color and transparency values that represent an image. An object of the **Bitmap** class is a subclass of the **DisplayObject** class, meaning that it can be placed on the display list. A **Bitmap** object encapsulates a **BitmapData** object.

Note: A 32-bit ARGB color value: I will have a lot more to say about pixels, color values, etc. in future lessons. My objective for this lesson is simply to get you to the point that you can create and display bitmap data.

A new object of the BitmapData class

The constructor for the Bitmap03 class begins in Listing 1. The constructor instantiates a new object of the **BitmapData** class with a red opaque background. The constructor parameters for a **BitmapData** object are:

- **width** :int -- The width of the bitmap image in pixels. The width is specified as 100 pixels in Listing 1.
- **height** :int -- The height of the bitmap image in pixels. The height is specified as 100 pixels in Listing 1.
- **transparent** -- Boolean (default = true) -- Specifies whether the bitmap image supports per-pixel transparency. The default value is true (transparent). A value of false (opaque) is specified in Listing 1.
- **fillColor** :uint (default = 0xFFFFFFFF -- A 32-bit ARGB color value that you use to fill the bitmap image area. The color red is specified in

Listing 1.

Draw a yellow cross on the background

A **BitmapData** object encapsulates a rectangular array of pixel data. The class provides several methods that make it possible for you to **get** and **set** individual pixel values in terms of color and transparency.

One of those methods is the method named **setPixel** that lets you replace *(or merge with transparency)* an existing pixel with a new pixel on the basis of the horizontal and vertical coordinates of the pixel.

Listing 2 uses a **for** loop to replace some of the red pixels in the **BitmapData** object with yellow pixels to form a yellow cross as shown in Figure 1.

Example:
Draw a yellow cross on the background.

```
      var yellow:uint = 0xFFFF00;

      for(var cnt:uint = 0; cnt < 100; cnt++){
        bitmapData.setPixel(cnt, cnt, yellow);
        bitmapData.setPixel(100 - cnt, cnt,
yellow);
      } //end for loop
```

The setPixel method

The **setPixel** method requires three parameters. The first two are the horizontal and vertical coordinates of the pixel whose color values are to be set. The third is an unsigned integer value that specifies the color that will be assigned to the pixel at that location.

Without getting into the details at this point, suffice it to say that the value assigned to the **yellow** variable in Listing 2 specifies the visible color of yellow.

Encapsulate the BitmapData object in a Bitmap object

The constructor for the **Bitmap** class has three parameters, but they all have default values. The first parameter, if provided, must be a reference to an object of type **BitmapData** .

Continuing with the constructor for the **Bitmap03** class, Listing 3 instantiates a new **Bitmap** object passing the **BitmapData** object prepared in Listing 2 as a parameter.

Example:
Encapsulate the BitmapData object in a Bitmap object and add it to the display.

```
      var bitmapObj:Bitmap = new
Bitmap(bitmapData);

      addChild(bitmapObj);

    } //end constructor
  } //end class
} //end package
```

Add the new Bitmap object to the display

Then Listing 3 calls the **addChild** method on the **Sprite** object to add the new **Bitmap** object to the display. This produces the screen output shown in Figure 1.

Simple enough?

This seems simple enough - right? However, there is a potential problem. In particular, an ActionScript project does not have access to any of those great Flex components. Therefore, if you are going to create ActionScript projects, you will need to have access to similar resources from some other source.

Not compatible with Flex

This is the form of the sample programs provided in the ActionScript documentation on the two web pages listed above . Unfortunately, I didn't see a warning on either of those pages to the effect that the programs are not compatible with Flex. I finally found a comment at the bottom of a different documentation page that gave me the first clue about the compatibility problem.

The program named Bitmap04

In this program, I will show you how to update the program named **Bitmap03** to make it compatible with Flex.

MXML code for the program named Bitmap04

We will begin with the MXML code shown in Listing 4 and repeated in Listing 9 for your convenience.

Example:
MXML code for the program named Bitmap04.

```
<?xml version="1.0" encoding="utf-8"?>

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>
```

```
</mx:Application>
```

Two files

This program consists of two files. One file, named **Bitmap04.mxml** contains the code shown in Listing 4. The other file named **Driver.as** contains the definition of a class named **Driver** . It is located in a package folder named **CustomClasses** , leading to the namespace declaration in Listing 4.

Instantiate an object of the Driver class

As you can see, the MXML code in Listing 4 simply instantiates an object of the **Driver** class. Beyond that point, the behavior of the program is completely controlled by ActionScript code.

Beginning of the Driver class for the program named Bitmap04

The **Driver** class begins in Listing 5. The code in Listing 5 is very similar to the code in Listing 1 with a few exceptions.

Example:
Beginning of the Driver class for the program named Bitmap04.

```
package CustomClasses{
  import mx.containers.VBox;
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import mx.core.UIComponent;

  public class Driver extends VBox{
    //Note that this class extends Sprite, nothing
shows
```

```
    // on the display. Therefore it extends VBox
instead.

    public function Driver(){
      //Prepare the display area. This is window
dressing.
      setStyle("backgroundAlpha",1.0);
      setStyle("backgroundColor",0x00FFFF);
      width = 150;
      height = 150;

      //Create a BitmapData object with a red
opaque
      // background.
      var bitmapData:BitmapData =
                new BitmapData(100, 100, false,
0xFF0000);
```

Extend the VBox class

This class extends **VBox** instead of extending **Sprite** as is the case in the previous program. Although this program will compile and run by extending **Sprite** , nothing appears on the output screen.

Apparently the reason is that Flex requires user interface display objects to be subclasses of the class named **UIComponent** , and the **Sprite** class is not a subclass of that class. This is the first major difference between this program and the previous one.

Some window dressing

The code at the beginning of the constructor in Listing 5 sets the height, width, background color, and transparency of the **VBox** container so that it can be seen in the display. As indicated in the comments, this is window dressing and is not a requirement for the program to execute properly. The cyan rectangle in Figure 2 is the **VBox** container.

A new BitmapData object

The last statement in Listing 5 instantiates a new **BitmapData** object identical to the one instantiated in the previous program.

Draw a yellow cross on the background

The code is listing 6 is identical to the code in Listing 2. This code draws the yellow cross on the red background that you see in Figure 2.

Example:
Draw a yellow cross on the background.

```
      var yellow:uint = 0xFFFF00;

      for(var cnt:uint = 0; cnt < 100; cnt++){
        bitmapData.setPixel(cnt, cnt, yellow);
        bitmapData.setPixel(100 - cnt, cnt, 
yellow);
      } //end for loop
```

Encapsulate the bitmap data in a Bitmap object and add it to the display

This is where the second major difference between the two programs arises. Although both programs encapsulate the bitmap data in the **Bitmap** object using the same code, the code required to add the **Bitmap** object to the output display is very different.

Example:
Encapsulate the bitmap data in a Bitmap object and add it to the display.

```
      var bitmapObj:Bitmap = new 
Bitmap(bitmapData);

      var uiComponent:UIComponent = new 
UIComponent();
      uiComponent.addChild(bitmapObj);

      addChild(uiComponent);
    } //end constructor

  } //end class
} //end package
```

Call the addChild method

In the previous ActionScript program, it was possible to simply call the **addChild** method to add the **Bitmap** object to the output display. However, that is not possible in a Flex application. If you try to do so, your program will throw a runtime error indicating an incompatible type.

Two requirements

In order to add a child to a **VBox** object, that child:

- Must be a subclass of the **DisplayObject** class and
- Must implement the **IUIComponent** interface.

While a **Bitmap** object is a subclass of **DisplayObject** , it does not implement the **IUIComponent** interface. Therefore, it is not compatible with being added directly to the **VBox** object.

Use a UIComponent object as an intermediary

One way to handle this is to add the **Bitmap** object to a new **UIComponent** object and then add the **UIComponent** object to the **VBox** object.

That is what I did in Listing 7. The result is shown in Figure 2 with the red **Bitmap** object contained in the cyan **VBox** object.

This is the second major difference between the two programs.

The end of the program

Listing 7 also signals the end of the class and the end of the program.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 8 through Listing 10. Use that code to create an ActionScript project and a Flex project. Compile and run the projects. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the source code for the two programs discussed in this lesson are provided below.

Example:
Source code for the program named Bitmap03.

```
/***************************************************
********
 * Bitmap03 11/29/09
 * Can be run as an ActionScript project. However,
some
 * changes are required to convert it to a Flex
project.
 *
 * For some reason, the size and position of the
100x100
 * pixel red square depends on the size and the
height to
 * width ration of the browser window.
***************************************************
*******/
package {
  import flash.display.Sprite
  import flash.display.Bitmap;
  import flash.display.BitmapData;

  public class Bitmap03 extends Sprite {

    public function Bitmap03(){
      //Create a BitmapData object with a red
opaque
      // background.
      var bitmapData:BitmapData =
                new BitmapData(100, 100, false,
0xFF0000);

      //Draw a yellow cross on the background
      var yellow:uint = 0xFFFF00;
      for(var cnt:uint = 0; cnt < 100; cnt++){
        bitmapData.setPixel(cnt, cnt, yellow);
        bitmapData.setPixel(100 - cnt, cnt,
yellow);
      } //end for loop
      //Encapsulate the bitmap data into a new
Bitmap
      // object and add it to the display. For
some
      // reason, the red square displays as 50x50
instead
      // of 100x100.
      var bitmapObj:Bitmap = new
Bitmap(bitmapData);
      addChild(bitmapObj);

    } //end constructor
  } //end class
} //end package
```

Example:
MXML code for the program named Bitmap04.

```
<?xml version="1.0" encoding="utf-8"?>
<!--Bitmap04
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Example:

Source code for the Driver class in the program named Bitmap04.

```
/*************************************************
********
 * Bitmap04 11/29/09
 * This is an update to Bitmap03 to make it
possible to
 * compile and run it as a Flex project. Unlike
Bitmap03
 * the red square produced by this program
displays at the
 * correct size of 100x100 pixels.
*************************************************
*******/
package CustomClasses{
  import mx.containers.VBox;
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import mx.core.UIComponent;

  public class Driver extends VBox{
    //Note that this class extends Sprite, nothing
shows
    // on the display. Therefore it extends VBox
instead.

    public function Driver(){
      //Prepare the display area. This is window
dressing.
      setStyle("backgroundAlpha",1.0);
      setStyle("backgroundColor",0x00FFFF);
      width = 150;
      height = 150;

      //Create a BitmapData object with a red
opaque
      // background.
      var bitmapData:BitmapData =
                new BitmapData(100, 100, false,
0xFF0000);

      //Draw a yellow cross on the background
      var yellow:uint = 0xFFFF00;
      for(var cnt:uint = 0; cnt < 100; cnt++){
        bitmapData.setPixel(cnt, cnt, yellow);
        bitmapData.setPixel(100 - cnt, cnt,
yellow);
      } //end for loop

      //Encapsulate the bitmap data into a new
Bitmap
      // object and add it to the display.
      var bitmapObj:Bitmap = new
Bitmap(bitmapData);

      //The following is necessary because the
Bitmap
      // class is not a subclass of UIComponent.
      var uiComponent:UIComponent = new
UIComponent();
      uiComponent.addChild(bitmapObj);
      addChild(uiComponent);
    } //end constructor

  } //end class
} //end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Bitmap Basics
- Files:
    - ActionScript0130\ActionScript0130.htm
    - ActionScript0130\Connexions\ActionScriptXhtml0130.htm

-end-

Fundamentals of Image Pixel Processing
Learn how to write skeleton code for creating a Bitmap object from an image file and how to display the bitmap. Learn two different ways to handle the image file, one of which results in a security issue that must be handled at compile time. Learn how to extract the BitmapData object from the Bitmap object and how to use the getPixels, setPixels, and setPixel32 methods to process the pixels in the bitmap. Learn how to apply a color inversion algorithm to invert the colors in a BitmapData object.

Note: Click Bitmap05 or Bitmap06 to run the ActionScript programs from this lesson. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming (OOP) with ActionScript.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

In an earlier lesson titled **Bitmap Basics** , I explained the differences between Flex projects and ActionScript projects. I also introduced you to the classes named **Bitmap** and **BitmapData** . Now it's time to put that knowledge to work. In this lesson, I will show you how to:

- Load the contents of an image file into a **BitmapData** object encapsulated in a **Bitmap** object.
- Use the **setPixel32** , **getPixels** , and **setPixels** methods to access and modify the color content of the individual pixels that make up an image.

Before getting into that, however, it will be useful to explain how image information is stored in files and in the computer's memory.

Vector graphics versus bitmap graphics

Graphics programming typically involves two main types of data: bitmaps and vector graphics. This lesson deals with bitmap data only. I will deal with vector graphics in a future lesson.

A rectangular array of pixels

When you take a picture with your digital camera, the scene is converted into a rectangular array containing millions of uniformly spaced colored dots. Those dots or picture elements *(pixels)* are stored on the memory card in your camera until you download them into your computer.

Width, height, and color depth

An image that is stored in this way is defined by the width and height of the array of pixels along with the number of bits that are used to define the color.

Up to a point, the more pixels that the camera produces to represent a given field of view, the better will be the image. Similarly, the more bits that are used to store the color, the better will be the overall quality of the image, particularly in terms of subtle shades of color.

The resolution

The number of pixels per unit area is commonly referred to as resolution. For example, the display monitor that I am currently using displays an array of 1280 x1024 pixels in a rectangular area with a diagonal measurement of 19 inches. (This is not a particularly high resolution monitor.)

The color depth

The number of bits used to represent the color of a pixel is commonly referred to as the color depth. Most modern computers routinely use a color depth of 32 bits. Note, however, that some file formats used for the storage and transmission of bitmap graphics data use fewer than 32 bits for the representation of each pixel in an image.

The RGB or ARGB color model

ActionScript supports a computer color model commonly known as the RGB model or the ARGB model. With this model, the color of each pixel is represented by three numeric color values and an optional transparency value *(alpha)* .

Each of the three color values represents the contribution or strength of a primary color: *red, green, and blue.* The final color of the pixel is a mixture of the primary colors. This is similar to a kindergarten student mixing red, green, and blue clay to produce a color that is different from red, green or blue. *(I don't have a clay analogy for transparency, however.)*

The effect of the transparency value

In some cases, the pixel also contains another value referred to as the *alpha* value *(ARGB)* that represents the transparency of the pixel.

Transparency comes into play when you draw a new image over an existing image. If the alpha value for a pixel is zero, there is no change in the color of the existing pixel because the new pixel is totally transparent. *(Although the effect is commonly referred to as transparency, the numeric value is actually proportional to opacity, which is the inverse of transparency.)*

Total opacity

If the alpha value indicates total opacity ***(often represented as either 1.0 or 255)*** , the color of the existing pixel is completely replaced by the color of the new pixel. *(I will explain the difference between 1.0 and 255 later.)*

Partial opacity

If the alpha value falls between 0 and 1.0 *(0 and 255)* , the colors of the existing pixel and the new pixel are combined to produce a new blended color. The result is as if you are viewing the original scene through colored glass.

An unsigned 32-bit chunk of memory

Typically, a pixel is represented in the computer by an unsigned 32-bit chunk of memory. Each of the three primary colors and the alpha value are represented by an eight-bit unsigned byte. The bytes are concatenated to form the 32-bit chunk of memory.

256 levels

This results in 256 levels of intensity for each of the primary colors along with 256 levels of transparency for the alpha byte. For example, if the red, green, and alpha bytes are equal to 255 and the blue byte is zero, the pixel will be displayed as bright yellow on a typical computer screen.

Bitmap image file formats

Different file formats are commonly used to store and transmit image data. It is usually desirable to reduce the size of the file required to store a given image while maintaining the quality of the image. This often results in a tradeoff between file size and image quality.

Different formats use different compression algorithms to reduce the size of the file. The bitmap image formats supported by Adobe Flash Player and Adobe Air are GIF, JPG, and PNG.

The GIF format

The Graphics Interchange Format *(GIF)* is a format that is often used to store low quality images in very small files. The format can store a maximum of 256 different colors and can designate one of those colors to represent a fully transparent pixel. By comparison, the typical ARGB format can represent more than sixteen million colors with 256 levels of transparency for each pixel.

The GIF format would not be very satisfactory for images produced by your digital camera, but it is fine for many purposes such as screen icons where high color quality is not an important consideration.

The JPEG format

This format, which is often written as JPG, was developed by the Joint Photographic Experts Group *(JPEG)* . This image format uses a lossy compression algorithm to allow 24-bit color depth with a small file size.

Lossy compression means that what comes out of the compressed file is not identical to what went in. The loss in picture quality is often acceptable, however, given that the format allows for different degrees of lossiness which is inversely related to the size of the compressed file. All of the digital cameras that I have owned produce JPEG files as the standard output and some of them allow the user to select the degree of compression and hence the degree of lossiness.

The JPEG format does not support alpha transparency. Therefore, it is not suitable as a file format for transmitting images with alpha data between computers.

The PNG format

Apparently there are some patent issues with the GIF format. The Portable Network Graphics *(PNG)* format was produced as an open-source alternative to the GIF file format.

The PNG format supports at least sixteen million colors and uses lossless compression. The PNG format also supports alpha transparency allowing for up to 256 levels of transparency in a compressed format.

The BitmapData class

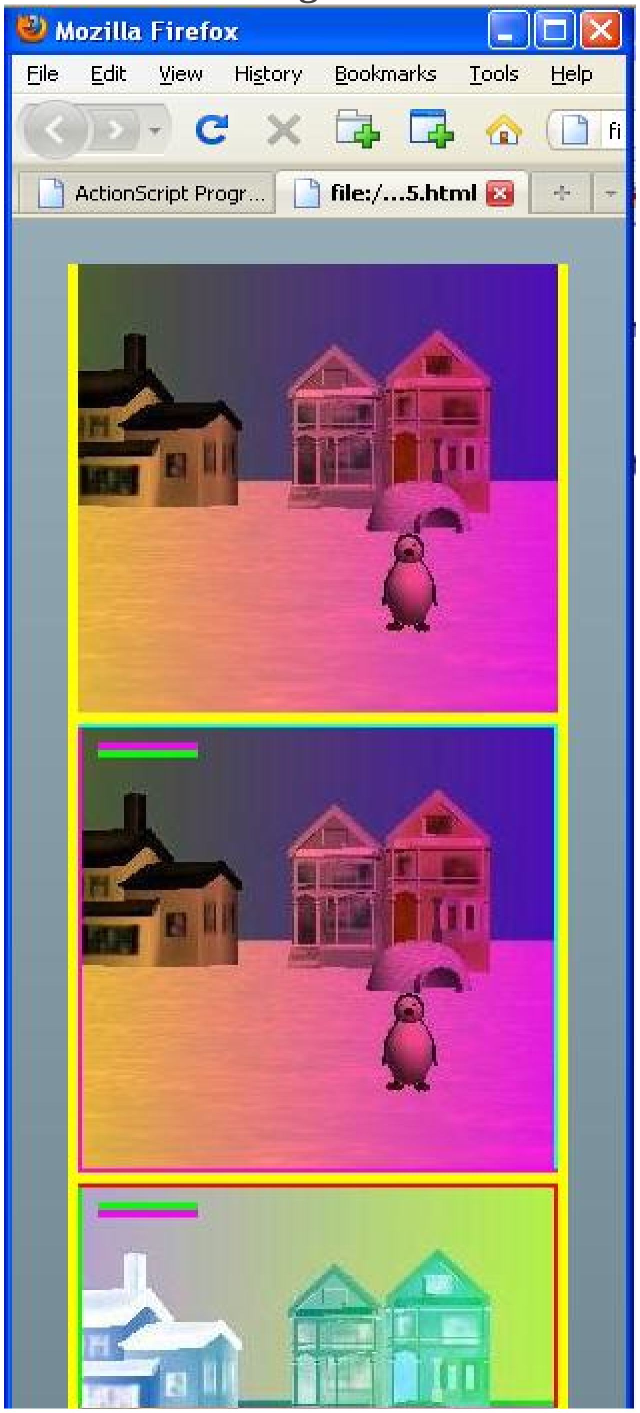
The **BitmapData** class in ActionScript 3 supports a 32-bit ARGB color model with more than sixteen million colors and 256 levels of alpha transparency per pixel.

Preview

I will explain two different programs in this lesson. One is named **Bitmap05** and the other is named **Bitmap06** . Both programs produce the same output, which is shown in Figure 1.

Screen output for both programs.

Figure



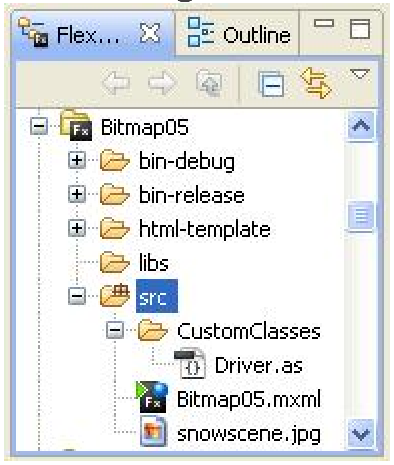
Screen output for both programs.

Program file structure

Figure 2 shows the program file structure taken from the Flex Builder 3 Navigator panel for the program named **Bitmap05** .

Program file structure for the program named Bitmap05.

Figure



```
Bitmap05
  bin-debug
  bin-release
  html-template
  libs
  src
    CustomClasses
      Driver.as
    Bitmap05.mxml
    snowscene.jpg
```

Program file structure for the program named Bitmap05.

The program file structure for **Bitmap06** is the same except for the name of MXML file.

I will use the programs named **Bitmap05** and **Bitmap06** to explain a variety of topics.

Skeleton code

First, I will provide skeleton code for creating a **Bitmap** object from an image file. You will learn how to use the skeleton code to create and display a bitmap from an image file as shown by the top image in Figure 1.

Extract the BitmapData object

I will show you how to extract the **BitmapData** object from the **Bitmap** object so that you can modify the pixels in the image. Once you are able to extract the **BitmapData** object, you will have the bitmap data exposed to the point that you can implement a variety of image processing algorithms such as the following:

- Smoothing
- Sharpening
- Edge detection
- Color filtering
- Color inversion
- Redeye correction

(For more information on how to implement image processing algorithms in general, see the tutorial lessons beginning with Lesson 340 in the section titled ***Multimedia Programming with Java*** *on my web site .)*

Using the getPixels, setPixels, and setPixel32 methods

I will explain how to use the **getPixels** , **setPixels** , and **setPixel32** methods for modifying the pixels in a bitmap as shown by the middle image in Figure 1. If you compare the middle image with the top image, you will see that two colored bars were added near the upper-left corner. In addition, colored borders were added to the middle image.

Color inversion

Finally, I will explain how to implement a color inversion algorithm as shown by the bottom image in Figure 1. If you compare the bottom image with the middle image, you will see that the colors of all of the pixels in the top half of the bottom image have been changed. The colors of the pixels were changed using a particular algorithm known widely as a color inversion algorithm.

The color inversion algorithm produces an output that is very similar to an old fashioned color film negative. The algorithm is economical to implement and totally reversible. Therefore, it is used by several major software products to highlight an image and show that the image has been selected for processing.

Discussion and sample code

As mentioned earlier, I will explain two different programs in this lesson. One is named **Bitmap05** and the other is named **Bitmap06** .

The program named Bitmap05

Will explain in fragments

I will explain the code for both programs in fragments. Both programs use the same MXML code but use different ActionScript code. Complete listings of the MXML file and the ActionScript files are provided in Listing 19 through Listing 21 near the end of the lesson.

MXML code for the program named Bitmap05

The MXML code for this program *(and the program named **Bitmap06** as well)* is shown in Listing 1 and also in Listing 19 for your convenience.

Example:
MXML code for the program named Bitmap05.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This program illustrates loading an image and
modifying the pixels in the image.
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Both programs are written almost entirely in ActionScript. As you can see, the MXML code instantiates a single object of the **Driver** class. Beyond that point, the behavior of both programs is controlled entirely by ActionScript code in the **Driver** class.

ActionScript code for the program named Birtmap05

A security issue

Let me begin by saying that I make no claims of expertise regarding security issues and the Flash Player.

No embedded image

I did not embed the image file shown in Figure 1 in the SWF file for the program named **Bitmap05** . Instead, I included it as a separate file in the *Release Build* of the program. As a result, it was necessary for me to change one of the XML elements in the following configuration file to make it possible for you to download and run the online version of the program.

C:\Program Files\Adobe\Flex Builder 3\sdks\3.2.0\frameworks\flex-config.xml

(Note that the configuration file may be in a different location on your computer.)

The required change

The required change was to set the value in the following element to false instead of true:

```
<use-network>false</use-network>
```

Why am I telling you this?

The ActionScript documentation seems to take for granted that you must modify the configuration file to handle the security issue. However, it took a very long time and a lot of searching for me to discover that in order to select certain compiler options, it is necessary to physically modify the configuration file shown above.

I knew generally the kind of change that was required, but I was expecting to find an option in the Flex Builder 3 IDE to allow me to change the compiler options on a project by project basis. If that capability exists in the IDE, I was unable to find it. *(Of course, once you know about the requirement, you can Google "flex-config.xml" and find hundreds of references to the topic.)*

I am telling you this in the hope that this information will save you countless hours of searching through the documentation to discover why you get a runtime error when you replicate this project and then try to download and run it in the Flash Player plugin.

Beginning of the ActionScript Driver class

The MXML code shown in Listing 1 instantiates an object of the **Driver** class. The ActionScript **Driver** class begins in Listing 2.

Example:
Beginning of the ActionScript Driver class.

```
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.display.Loader;
  import flash.events.*;
  import flash.geom.Rectangle;
  import flash.net.URLRequest;
  import flash.utils.ByteArray;

  import mx.containers.VBox;
  import mx.controls.Image;
  import flash.system.Security;

//==================================================
====//

  public class Driver extends VBox {
    private var url:String =
"snowscene.jpg";//image file

    public function Driver(){//constructor
      //Make the VBox visible.
      setStyle("backgroundColor",0xFFFF00);
      setStyle("backgroundAlpha",1.0);
```

Establish the name and path of the image file

The class definition in Listing 2 begins by declaring and populating an instance variable named **url** with the name of the image file shown in Figure 2. As you can see in Figure 2, this file was located in the same folder as the MXML file. Therefore, no path information was required to specify the location of the image file.

The constructor

The constructor for the class also begins in Listing 2. This class extends the **VBox** class. The code in Listing 2 causes the background color of the **VBox** object to be yellow and also causes the yellow background to be completely opaque. You can see the opaque yellow background of the **VBox** object in Figure 1.

Opacity is often represented as either 1.0 or 255

Remember that I told you earlier that the opacity value is *"often represented as either 1.0 or 255."* Usually when you set the alpha value as a property of a Flex component, you must specify a value ranging from 0.0 to 1.0 with a value of 1.0 being completely opaque. On the other hand, when you are dealing with the actual alpha value in bitmap data, you must specify a value ranging from 0 to 255 with a value of 255 being completely opaque.

Start the process of loading the image file

Listing 3 starts the process of loading the image file. As I mentioned earlier, this program does not embed the image file in the SWF file. *(The program named **Bitmap06** , which I will explain later, does embed the image file in the SWF file.)* Instead, the image file for this program ends up as a separate file on the server that must be downloaded in addition to the SWF file. *(The*

ActionScript literature contains numerous discussions regarding the pros and cons of embedding versus not embedding resource files.)

Example:
Start the process of loading the image file.

```
      var loader:Loader = new Loader();

      //Register event listeners on the load 
process
      loader.contentLoaderInfo.addEventListener(

Event.COMPLETE,completeHandler);
      loader.contentLoaderInfo.addEventListener(

IOErrorEvent.IO_ERROR,ioErrorHandler);

      var request:URLRequest = new 
URLRequest(url);
      loader.load(request);
```

Straightforward code

The code in Listing 3 is fairly straightforward. You should be able to understand it if you analyze it using the ActionScript documentation.

The Event.COMPLETE event handler

The main thing that I want to emphasize from Listing 3 is the registration of the **Event.COMPLETE** event handler. Note that this event handler is registered on the **contentLoaderInfo** property of the **Loader** object and not on the **Loader** object itself. The documentation has this to say about this property:

"Returns a LoaderInfo object corresponding to the object being loaded. LoaderInfo objects are shared between the Loader object and the loaded content object. The LoaderInfo object supplies loading progress information and statistics about the loaded file.

Events related to the load are dispatched by the LoaderInfo object referenced by the contentLoaderInfo property of the Loader object. The contentLoaderInfo property is set to a valid LoaderInfo object, even before the content is loaded, so that you can add event listeners to the object prior to the load."

The Event.COMPLETE event

The documentation has this to say about the **Event.COMPLETE** event:

"Dispatched when data has loaded successfully. In other words, it is dispatched when all the content has been downloaded and the loading has finished. The complete event is always dispatched after the init event. The init event is dispatched when the object is ready to access, though the content may still be downloading."

Beginning of the complete event handler

The **complete** event handler that is registered in Listing 3 begins in Listing 4. This handler is executed when the load process is complete and the image data is available.

Example:
Beginning of the complete event handler.

```
    private function
completeHandler(event:Event):void{

      //Get, cast, and save a reference to a
Bitmap object
      // containing the content of the image file.
      var original:Bitmap = Bitmap(

event.target.loader.content);

      //Set the width and height of the VBox
object based
      // on the size of the original bitmap.
      this.width = original.width + 10;
      this.height = 3*original.height + 12;
```

Get a reference to the Bitmap object

The handler begins by using the incoming reference to the **Event** object to execute a complex statement that ends up with a reference to the **Bitmap** object. However, that reference is received as type **DisplayObject** and must be cast to type **Bitmap** to be used for its intended purpose in this program. The reference is cast to type **Bitmap** and saved in the variable named **original** .

When the first statement in Listing 4 finishes executing, the variable named **original** refers to a **Bitmap** object containing the image from the image file specified in Listing 2.

Set the dimensions of the VBox object

After creating the new **Bitmap** object, Listing 4 uses the dimensions of the **Bitmap** object to set the dimensions of the **VBox** object, which is shown by the yellow background in Figure 1.

Can use almost any image

All of the placement information for the images shown in Figure 1 is based on the dimensions of the **Bitmap** object. Therefore, you should be able to substitute any JPEG, PNG, or GIF image file in place of my file so long as the name of the file matches the name and location of the file specified in

Listing 2. Note however that your image file will need to be wide enough and tall enough to prevent the magenta and green color bars added to the center image in Figure 1 from extending outside the image.

Dealing with a type compatibility issue

In the earlier lesson titled **Bitmap Basics** , I explained that in order to add a child to a **VBox** object, that child:

- Must be a subclass of the **DisplayObject** class and
- Must implement the **IUIComponent** interface.

While a **Bitmap** object is a subclass of **DisplayObject** , it does not implement the **IUIComponent** interface. Therefore, it is not compatible with being added directly to the **VBox** object. I resolved the issue in that lesson by encapsulating the **Bitmap** object in an object of the **UIComponent** class, which implements the **IUIComponent** interface.

Encapsulate the Bitmap in an Image object

In this lesson, I decided to be more specific and encapsulate the **Bitmap** object in an object of the **Image** class. This is allowable because the **Image** class is a subclass of the **UIComponent** class.

Listing 5 encapsulates the **Bitmap** in an **Image** object and adds it to the **VBox** object to be displayed at the top of and five pixels to the right of the left edge of the **VBox** as shown by the top image in Figure 1.

Example:
Encapsulate the Bitmap in an Image object.

```
      //Encapsulate the bitmap in an Image object
and add
      // the Image object to the VBox. Display it
at
      // x=5 and y=0
      original.x = 5;
      original.y = 0;
      var imageA:Image = new Image();
      imageA.addChild(original);
      this.addChild(imageA);
```

A curious situation

This brings up a curious situation regarding the placement of the **Image** objects in the **VBox** object. Normally, if you instantiate **Image** objects and populate them directly from the contents of image files ***(by calling the load method on the Image object)*** , you can add them to a **VBox** object without the requirement to specify the locations at which the images will be displayed. The layout management rules of the **VBox** object determine how they are displayed.

This case is different

In this case, however, if you instantiate **Image** objects and populate them with **Bitmap** objects by calling the **addChild** method as shown in Listing 5, you must specify the display locations of the **Bitmap** objects within the **VBox** object. If you don't, they all end up in the upper-left corner of the **VBox** .

Honoring the boundaries of the VBox object

Also, if you specify the dimensions of the **VBox** object and add more images of the first type than the size of the **VBox** object can accommodate, scroll bars automatically appear on the **VBox** object.

In this case, however, if you specify the locations such that the **Image** objects won't all fit within the boundaries of the **VBox** object, the images extend outside the bounds of the **VBox** object.

I will leave it as an exercise for the student to sort through all of that.

Clone the original Bitmap to create a duplicate Bitmap

We have now reached the point where we could access the **BitmapData** object encapsulated in the **Bitmap** object and modify the pixel data that comprises the image. However, instead of modifying the pixels in the original **Bitmap** , I elected to create a duplicate bitmap and modify the pixels in the duplicate. That makes it possible to compare the unmodified image *(top image in Figure 1)* with the modified image *(middle image in Figure 1).*

Listing 6 calls the **clone** method on the original **Bitmap** object to create a duplicate **Bitmap** object, and saves the duplicate **Bitmap** object's reference in the variable named **duplicateB** .

Example:
Clone the original Bitmap to create a duplicate Bitmap.

```
      var duplicateB:Bitmap = new Bitmap(

original.bitmapData.clone());

      duplicateB.x = 5;
      duplicateB.y = original.height;

      var imageB:Image = new Image();
      imageB.addChild(duplicateB);
      this.addChild(imageB);
```

Display the duplicate bitmap

Then Listing 6 adds the duplicate bitmap to a new **Image** object, positions the duplicate bitmap immediately below the top image in Figure 1 and adds the new image to the **VBox** . This is the middle image in Figure 1.

Another curious circumstance

Curiously, the middle image in Figure 1 is five or six pixels further down than I expected it to be. This produces a gap of five or six pixels between the top two images in Figure 1. I am unable to explain the reason for the gap at this time, but I suspect that it may have something to do with the layout rules of the **VBox** container object. When I place two or more ordinary **Image** objects in a **VBox** container, they appear in a vertical stack separated by about five or six pixels. However, that is total speculation on my part.

Modify the duplicate Bitmap

Listing 7 calls the **modify** method passing a reference to **duplicateB** as a parameter. This causes the middle image in Figure 1 to be modified in two different ways.

- First, the magenta and green rows of pixels are inserted near the upper left corner.
- Then a colored border two pixels thick is inserted around the four edges of the bitmap.

Example:
Modify the duplicate Bitmap.

```
      modify(duplicateB);
```

Explain the modify method

At this point, I will put the explanation of the **complete** event handler on hold and explain the method named **modify** .

Beginning of the modify method

The **modify** method begins in Listing 8.

Example:
Beginning of the modify method.

```
    private function modify(bitmap:Bitmap):void{

      var bitmapData:BitmapData = 
bitmap.bitmapData;
```

The incoming **Bitmap** object encapsulates a **BitmapData** object, which is referenced by a property of the **Bitmap** object named **bitmapData** . Listing 8 gets a copy of that reference and saves it in a local variable named **bitmapData** .

Process pixels using the getPixels and setPixels methods

Listing 9 begins by instantiating a new empty object of type **ByteArray** .

Example:
Process pixels using the getPixels and setPixels methods.

```
      var rawBytes:ByteArray = new ByteArray();
      rawBytes = bitmapData.getPixels(
                              new 
Rectangle(10,10,50,8));
```

The ByteArray class

According to the documentation ,

"The ByteArray class provides methods and properties to optimize reading, writing, and working with binary data."

A **ByteArray** object is an object that can be used to store and access data using either square bracket notation [] or method calls. The main benefit of using this data structure from our viewpoint is that it will decompose the 32-bit integers into 8-bit bytes and allow us access the pixel data one byte at a time. Otherwise it would be necessary for us to perform the decomposition ourselves using bit shift operations.

The getPixels method

Listing 9 populates the **ByteArray** object by calling the **getPixels** method on the **BitmapData** object. According to the documentation , this method

*"Generates a byte array from a **rectangular region** of pixel data. Writes an unsigned integer (a 32-bit unmultiplied pixel value) for each pixel into the byte array."*

A Rectangle object

A new **Rectangle** object is instantiated to define the *rectangular region* from which the pixels are extracted. According to the documentation, the constructor for this class

*"Creates a new **Rectangle** object with the top-left corner specified by the x and y parameters and with the specified width and height parameters."*

The rectangular region specified by the parameters in Listing 9 has its upper-left corner at (10,10) , is 50 pixels wide, and 8 pixels high. This is the rectangular occupied by the magenta and green horizontal bars near the upper-left corner of the middle image in Figure 1.

The ByteArray object is populated with pixel data

When the **getPixels** method returns in Listing 9, the pixels from that rectangular region are stored in the **ByteArray** object referred to by

rawBytes .

The organization of the pixel data

The first four bytes in the array belong to the pixel in the upper-left corner of the rectangular region. The next four bytes belong to the pixel immediately to the right of that one. The array is populated by the data from the rectangular region on a row by row basis.

Each set of four bytes represent one pixel in ARGB format. In other words, the first byte in the four-byte group is the alpha byte. That byte is followed by the red byte, the green byte, and the blue byte in that order. This information is critical when time comes to use the data in the array to modify the pixel data.

The general procedure

The general procedure when using this approach is to extract a rectangular region of pixels into the array, modify the array data, and then call the **setPixels** method to write the modified color data back into the area of memory that represents the rectangular region from the bitmap data.

A very useful format

The format of the data in the **ByteArray** object is very useful when you need to modify consecutive pixels on a row by row basis. It is less useful, but can be used when you need to modify pixels whose locations are more random in nature.

In our case, we want to set the color of all the pixels in the top four rows of the rectangular region to magenta and we want to set the color of all the pixels in the bottom four rows of the rectangular region to green as shown by the middle image in Figure 1. This data format is ideal for that kind of operation.

Modify the pixels in the rectangular region

Listing 10 sets the colors of the pixels in the top four rows to magenta and sets the colors of the pixels in the bottom four rows to green without modifying the value of the alpha byte.

Example:
Modify the pixels in the rectangular region.

```
      var cnt:uint = 1;
      while(cnt < rawBytes.length){
        if(cnt < rawBytes.length/2){
          rawBytes[cnt] = 255;
          rawBytes[cnt + 1] = 0;
          rawBytes[cnt + 2] = 255;
        }else{
          rawBytes[cnt] = 0;
          rawBytes[cnt + 1] = 255;
          rawBytes[cnt + 2] = 0;
        } //end if-else

        cnt += 4;//Increment the counter by 4.
      }//end while loop
```

The magenta and green color values

A magenta pixel is produced by setting the red and blue color bytes to full strength *(255)* and setting the green color byte to 0. A green pixel is produced by setting the red and blue color bytes to 0 and setting the green color byte to 255.

You should be OK by now

Knowing what you now know, you should have no difficulty understanding how the data in the **ByteArray** object is modified to produce the magenta and green colored areas near the upper-left corner of the middle image of Figure 1.

Not quite finished yet

Note, however that we haven't modified the actual pixel data in the bitmap yet. So far we have made a copy of all the pixel data in the rectangular region and have modified the color values in the copy of the pixel data. We still need to write the modified pixel data back into the **BitmapData** object to actually modify the image.

Put the modified pixel data back into the same rectangular region

Listing 11 calls the **setPixels** method to store the pixel data that is contained in the **rawBytes** array back into the same rectangular region of the bitmap image.

Example:
Put the modified pixel data back into the same rectangular region.

```
      rawBytes.position = 0;//this is critical
      bitmapData.setPixels(
                      new
Rectangle(10,10,50,8),rawBytes);
```

The position property of the ByteArray object

With one exception, you should have no difficulty understanding the code in Listing 11. That exception has to do with the **ByteArray** property named **position** . Here is what the documentation has to say about the position property:

"Moves, or returns the current position, in bytes, of the file pointer into the ByteArray object. This is the point at which the next call to a read method

starts reading or a write method starts writing."

Whether or not you understand what that means, it is critical that you set the value of the **position** property to zero before calling the **setPixels** method to cause all of the data in the array to be written into the **BitmapData** object. Otherwise, you will get a runtime error.

Some things worth noting

A couple of things are worth noting. First, there was no technical requirement to write the data from the array back into the same rectangular region from which it was read. It could have been written into a different rectangular region in the same bitmap, it could have been written into several different rectangular regions, or it could even have been written into a completely different **BitmapData** object.

No requirement to read the bitmap data

Second, since the code in Listing 10 stored color data into the array that was totally independent of the color values in the **BitmapData** object, there was no requirement to read the color data from the **BitmapData** object in the first place. We could simply have instantiated a new **ByteArray** object and set its length to the product of the width and the height of the rectangular region. Then we could have executed the code in Listing 10 to populate the bytes in the array with magenta and green color values. Then we could have executed the code in Listing 11 to write the pixel data into an appropriate rectangular region in the **BitmapData** object.

On the other hand...

On the other hand, had we wanted to do something like emphasize the green color and deemphasize the blue color in the rectangular region, we would have needed to call the **getPixels** method to get the actual pixel data from the **BitmapData** object into the array. Having that pixel data available, we could have:

- Multiplied the green color value in each pixel by 1.2,
- Multiplied the blue color values in each pixel by 0.8,

- Called the **setPixels** method as in Listing 11 to store the modified pixel data back into the same rectangular region of the **BitmapData** object.

Process pixels using the setPixel32 method

The code in Listing 12 uses the **setPixel32** method to put a magenta border on the left edge of the bitmap and a cyan border on the right edge of the **BitmapData** object as shown in the middle image in Figure 1. *(The border is two pixels thick.)*

Example:
Process pixels using the setPixel32 method.

```
      for(var row:uint = 0;row <
bitmapData.height;

row++){
        bitmapData.setPixel32(0,row,0xFFFF00FF);
        bitmapData.setPixel32(1,row,0xFFFF00FF);
        bitmapData.setPixel32(bitmapData.width -
1,
                                      row,0xFF00FFFF);
        bitmapData.setPixel32(bitmapData.width -
2,
                                      row,0xFF00FFFF);
      }//end for loop
```

The setPixel32 method and its cousins

The **setPixel32** method and its cousin the **setPixel** method, along with the **getPixel32** method and the **getPixel** method, are completely different from the **getPixels** method and the **setPixels** method used earlier.

And the differences are...

Each call to the **getPixels** method or the **setPixels** method deals with all of the pixels in a specified rectangular region.

Each call to the **getPixel** method, the **getPixel32** method, the **setPixel** method, or the **setPixel32** method deals with only one pixel. That pixel is identified by the horizontal and vertical coordinates of a single pixel in the **BitmapData** object.

Getting and/or setting a pixel value

As you have probably guessed by now, the **getPixel** and **getPixel32** methods are used to return the value of a single pixel from the specified location. Both of these methods return a 32-bit data value of type **uint** .

Similarly, the **setPixel** and **setPixel32** methods are used to write a 32-bit unsigned integer value into a specified location in the **BitmapData** object.

Not decomposed into separate bytes

Unlike with the **getPixels** method used with the **ByteArray** object, the **getPixel** and **getPixel32** methods don't decompose the 32-bit integer value into separate bytes for alpha, red, green, and blue. If you need to separate the returned value into individual bytes, you must accomplish that yourself.

The order of the bytes in the returned value is ARGB. In other words, the leftmost eight bits contain the alpha value, the rightmost eight bits contain the blue value, and the red and green bytes are in the middle.

The difference between the methods

The difference between the methods with 32 in the name the methods without 32 in the name has to do with the alpha byte. The two methods without 32 in the name return a 32-bit unsigned integer but only the 24 RGB bits are meaningful. The eight alpha bits are not meaningful. On the other hand, for the methods with 32 in the name, all four bytes including the alpha byte are meaningful.

Set the pixels to create a border

The code in Listing 12 uses a **for** loop and the **setPixel32** method to set the pixel color to fully opaque magenta *(red plus blue)* for the first two pixels in each row of pixels and to set the pixel color to fully opaque cyan *(green plus blue)* for the last two pixels in each row. This produces a magenta border with a thickness of two pixels on the left edge of the middle image in Figure 1 and produces a cyan border with a thickness of two pixels on the right edge of the middle image in Figure 1.

Put borders on the top and bottom edges

Listing 13 uses similar code to put a cyan border along the top edge and a magenta border along the bottom edge of the middle image in Figure 1.

Example:
Put borders on the top and bottom edges.

```
      for(var col:uint = 0;col < bitmapData.width;

col++){
        bitmapData.setPixel32(col,0,0xFF00FFFF);
        bitmapData.setPixel32(col,1,0xFF00FFFF);

bitmapData.setPixel32(col,bitmapData.height - 1,
                              0xFFFF00FF);

bitmapData.setPixel32(col,bitmapData.height - 2,
                              0xFFFF00FF);

      } //End for loop

    } //end modify method
```

The end of the modify method

Listing 13 also signals the end of the **modify** method. When the method returns from the call that was made in Listing 7, the pixels in the red and green rectangular region near the upper-left corner of the middle image of Figure 1 have been modified relative to the original image shown in the top image in Figure 1. In addition, the pixels along all four edges of the middle image have been replaced by magenta and cyan pixels to produce a border with a thickness of two pixels.

Return to the complete event handler

Returning now to where we left off in the **complete** event handler in Listing 7, the code in Listing 14:

- Creates another duplicate **BitmapData** object.
- Encapsulates it in an **Image** object.
- Places it at the bottom of Figure 1.
- Calls the **modify** method to modify the pixels just like the middle image in Figure 1.
- Calls the **invert** method to invert the colors of all the pixels in the top half of the **BitmapData** object to produce the final image shown at the bottom of Figure 1.

Example:
Return to the complete event handler.

```
      //Clone the original bitmap to create
another
      // duplicate.
      var duplicateC:Bitmap = new Bitmap(

original.bitmapData.clone());
      //Place the duplicateC below the other two
in the
      // VBox.
      duplicateC.x = 5;
      duplicateC.y = 2*original.height;

      var imageC:Image = new Image();
      imageC.addChild(duplicateC);
      this.addChild(imageC);

      //Modify the pixels as above to add some
color to
      // the image.
      modify(duplicateC);
      //Now invert the colors in the top half of
this
      // bitmap. Note that the magenta and green
colors
      // swap positions.
      invert(duplicateC);

    } //end completeHandler
```

Color inversion

The color inversion algorithm is

- Very fast to execute.
- Totally reversible.
- Guaranteed to convert every pixel to a different color.

Usually the new color is readily distinguishable from the old color.

A comparison

If you compare the top half of the bottom image in Figure 1 with the top half of the other two images, you can see the dramatic effect of color inversion. However, all that is required to exactly restore the original colors is to run the inverted color pixels through the inversion process again.

Because of these characteristics, some major software products use color inversion to change the colors in an image that has been selected for processing to provide a visual indication that it has been selected.

The color inversion algorithm

To invert the color of a pixel, you simply subtract the red, green, and blue color values from 255 without modifying the alpha value. To reverse the process, you simply subtract the inverted color values from 255 again, which produces the original color values.

Beginning of the invert method

The **invert** method begins in Listing 15.

Example:
Beginning of the invert method.

```
    private function invert(bitmap:Bitmap):void{
      //Get the BitmapData object.
      var bitmapData:BitmapData =
bitmap.bitmapData;

      //Get a one-dimensional byte array of pixel
data
      // from the top half of the bitmapData
object
      var rawBytes:ByteArray = new ByteArray();
      rawBytes = bitmapData.getPixels(new
Rectangle(

0,0,bitmapData.width,bitmapData.height/2));
```

The code in Listing 15 gets the **BitmapData** object encapsulated in the incoming **Bitmap** object and extracts the pixel data from a rectangle that comprises the entire top half of the bitmap data into a **ByteArray** object.

Apply the inversion algorithm

Listing 16 applies the color inversion algorithm to all of the pixel data in the **ByteArray** object by subtracting each color value from 255 and storing the result back into the same element of the **ByteArray** object.

Example:
Apply the inversion algorithm.

```
      var cnt:uint = 1;
      while(cnt < rawBytes.length){
        rawBytes[cnt] = 255 - rawBytes[cnt];
        rawBytes[cnt + 1] = 255 - rawBytes[cnt +
1];
        rawBytes[cnt + 2] = 255 - rawBytes[cnt +
2];

        cnt += 4;//increment the counter
      }//end while loop
```

Put the modified pixel data back into the BitmapData object

Listing 17 calls the **setPixels** method to put the modified pixel data back into the **BitmapData** object producing the final output shown in the bottom image of Figure 1.

Example:
Put the modified pixel data back into the BitmapData object.

```
      rawBytes.position = 0;//this is critical
      bitmapData.setPixels(new Rectangle(

0,0,bitmapData.width,bitmapData.height/2),
                rawBytes);

    } //end invert method
    //--------------------------------------------
------//

  } //end class
} //end package
```

This is a case where the new pixel color values depend on the original color values. Therefore, it was necessary to get and use the old color values to compute the new color values.

An interesting side note

If you compare the two color bars in the upper-left corner of the middle and bottom images in Figure 1, you will see that they appear to have swapped positions. This is because the numeric value of magenta is the inverse of the numeric value of cyan and vice versa. The same is true of the cyan and magenta borders.

The end of the program

Listing 17 also signals the end of the program named **Bitmap05** .

The program named Bitmap06

Behaves just like Bitmap05

This program behaves just like the program named Bitmap05. Therefore, the screen output shown in Figure 1 applies to this program as well as to the program named **Bitmap05** .

The difference between the two programs

This program differs from the program named **Bitmap05** in terms of how and when the image file is loaded. With the program named **Bitmap05** , the image file was maintained as a separate file and downloaded with the SWF file. Then it was loaded at runtime. This resulted in the security issue discussed earlier.

This program embeds the image file into the SWF file at compile time. Since there is no image file to be loaded from the local file system at runtime, the security issue does not apply to this program.

Code modifications

As you might imagine, it was necessary to make some modifications to the program code to accomplish this difference. I will explain those modifications in the following paragraphs.

MXML code for the program named Bitmap06

The MXML code for this program is the same as shown in Listing 19 for the program named **Bitmap05** .

ActionScript code for the program named Bitmap06

I will present and explain only the code that is different from the program named **Bitmap05** . However, a complete listing of the code for this program is provided in Listing 21 near the end of the lesson.

Code that is different in the program named Bitmap06

Listing 18 shows the code that is different in the program named **Bitmap06** . Since some of the code is the same, I have highlighted the code that is different with comments.

Example:
Code that is different in the program named Bitmap06.

```
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.geom.Rectangle;
  import flash.utils.ByteArray;
  import mx.containers.VBox;
  import mx.controls.Image;
  import mx.events.FlexEvent//different

//==================================================
====//

  public class Driver extends VBox {
    private var image:Image = new
Image();//different

    public function Driver(){//constructor
      //Make the VBox visible.
      setStyle("backgroundColor",0xFFFF00);
      setStyle("backgroundAlpha",1.0);

      [Embed("snowscene.jpg")]//different
      var img:Class;//different

      image.load(img);//different

      //Note that the type of completion event
specified
```

```
      // here is different from the type of
completion
      // event used in Bitmap05.
      //Different

this.addEventListener(FlexEvent.CREATION_COMPLETE,

completeHandler);
    } //end constructor
    //--------------------------------------------
------//

    //This handler method is executed when the
VBox has
    // been fully created. Note that the type of
the
    // incoming parameter is more specific than
was the
    // case in Bitmap05. However, it isn't used in
this
    // program.
    //Different
    private function completeHandler(

event:mx.events.FlexEvent):void{
      //Get and save a reference to a Bitmap
object
      // containing the content of the image file.
This
      // statement is different from Bitmap05.
      //Different
      var original:Bitmap = Bitmap(image.content);

      //Everything beyond this point is identical
to
      // Bitmap05 except that the IO error handler
was
```

```
      // removed. It isn't needed for an embedded
image
      // file.
```

A new import directive

The differences begin in Listing 18 with an import directive for the class named **FlexEvent** . This class was not needed in the program named Bitmap05.

Instantiation of an Image object

The differences continue in Listing 18 with the declaration and instantiation of an object of the class **Image** . The embedded image file will be loaded into this object at runtime. Then the **Bitmap** object encapsulated in the **Image** object will be extracted and passed to the **modify** method and the **invert** method the same as before.

Embedding the image file

The two lines of code beginning with the word **[Embed** provide the mechanism by which the image file is embedded into the SWF file. The first line specifies the name and path to the image file. In this case, it was in the same folder as the MXML file so no path was required.

The strange syntax of the **Embed** code means that it really isn't an executable programming statement. Instead, it is an instruction to the compiler telling the compiler to embed the file in the SWF file.

Declare a variable to refer to the embedded file

The line immediately following the **Embed** directive declares a variable of type **Class** named **img** that can later be used to refer to the embedded file.

Load the file contents into the Image object

The second line of code following the **Embed** directive causes the contents of the embedded image file to be loaded into the **Image** object at runtime. Note that the embedded image file is referenced by the variable named **img** that was declared along with the **Embed** directive and passed as a parameter to the load method.

No need to worry about IO errors at runtime

Because the image file is read at compile time and embedded into the SWF file, there is no need to provide an IO error handler that will be executed as a result of a runtime IO error involving the image file. If there is a problem reading the file, that problem will occur when the program is compiled and the SWF file is written.

Register a creationComplete event listener on the VBox

The last statement in the constructor registers a **creationComplete** event handler on the **VBox** . This is considerably different from the program named **Bitmap05** . First, the event handler is registered on the **VBox** instead of being registered on **loader.contentLoaderInfo** . Second, the type of the completion event is different between the two programs.

The **creationComplete** event will be

"Dispatched when the component (***VBox*** *) has finished its construction, property processing, measuring, layout, and drawing."*

The assumption is that by the time the event is dispatched, the bitmap data will have been successfully loaded into the **Image** object.

Differences in the creationComplete event handler

The first difference in the complete event handler is the type of event passed to the handler. The type **FlexEvent** shown in Listing 18 is more specialized than the type **Event** shown in Listing 4. However it doesn't matter in this case because the incoming reference to the event object isn't used.

Getting the Bitmap object

The **Bitmap** object in an **Image** object is referred to by the property named **content** .

The last statement that is marked as being different gets a reference to the **Bitmap** object and stores it in the variable named **original** just like was done in Listing 4.

As before, referencing the **content** property returns the **Bitmap** object as type **DisplayObject** . Therefore, it must be cast to type **Bitmap** before it can be used for the intended purpose of this program.

Beyond this point - no changes

Beyond this point, the two programs are identical except that the IO error handler was omitted from this program. As I explained earlier, because the image file is embedded in the SWF file at compile time, there is no need to worry about IO errors involving the image file at runtime.

Run the programs

I encourage you to run the online versions of the two programs from the web. Then copy the code from Listing 19 through Listing 21. Use that code to create Flex projects. Compile and run the projects. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the MXML code and the ActionScript code for the programs discussed in this lesson are provided below.

Example:
MXML code for the program named Bitmap05.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This program illustrates loading an image and
modifying the pixels in the image.
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Example:
ActionScript code for the program named Bitmap05.

```
/*Bitmap05
Provides skeleton code for creating a Bitmap
object from
an image file.
Explains the use of the getPixels, setPixels, and
```

Example:

ActionScript code for the program named Bitmap06.

```
/*Bitmap06
This is an update to Bitmap05 that uses an image
that is
embedded in the SWF file rather than a separate
downloaded image file. This eliminates the
requirement
to make the following change to the configuration
file
at:

C:\Program Files\Adobe\Flex Builder 3\sdks\3.2.0\
frameworks\flex-config.xml

   <!-- Prevents SWFs from accessing the network.
-->
   <use-network>false</use-network>

The behavior of this program is identical to the
behavior of Bitmap05.
**************************************************
*******/
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.geom.Rectangle;
  import flash.utils.ByteArray;
  import mx.containers.VBox;
  import mx.controls.Image;
  import mx.events.FlexEvent;

//==================================================
====//

  public class Driver extends VBox {
    private var image:Image = new Image();
    public function Driver(){//constructor
      //Make the VBox visible.
      setStyle("backgroundColor",0xFFFF00);
      setStyle("backgroundAlpha",1.0);

      [Embed("snowscene.jpg")]
      var img:Class;
      image.load(img);

      //Note that the type of completion event
specified
      // here is different from the type of
completion
      // event used in Bitmap05.

this.addEventListener(FlexEvent.CREATION_COMPLETE,

completeHandler);
    } //end constructor
    //--------------------------------------------------
------//

    //This handler method is executed when the
VBox has
    // been fully created. Note that the type of
the
    // incoming parameter is more specific than
was the
    // case in Bitmap05. However, it isn't used in
this
    // program.
    private function completeHandler(

event:mx.events.FlexEvent):void{
      //Get and save a reference to a Bitmap
object
```

```
      // containing the content of the image file.
This
      // statement is different from Bitmap05.
      var original:Bitmap = Bitmap(image.content);

      //Everything beyond this point is identical
to
      // Bitmap05 except that the IO error handler
was
      // removed. It isn't needed for an embedded
image
      // file.

      //Set the width and height of the VBox
object based
      // on the size of the original bitmap.
      this.width = original.width + 10;
      this.height = 3*original.height + 12;

      //Encapsulate the bitmap in an Image object
and add
      // the Image object to the VBox. Display it
at
      // x=5 and y=0
      original.x = 5;
      original.y = 0;
      var imageA:Image = new Image();
      imageA.addChild(original);
      this.addChild(imageA);

      //Clone the original bitmap to create a
duplicate.
      var duplicateB:Bitmap = new Bitmap(

original.bitmapData.clone());
      //Place the duplicate bitmap below the
original in
```

```
      // the VBox. There is a six-pixel downward
shift
      // that I am unable to explain at this time.
The
      // shift produces a gap of about six pixels
between
      // the two images.
      duplicateB.x = 5;
      duplicateB.y = original.height;

      var imageB:Image = new Image();
      imageB.addChild(duplicateB);
      this.addChild(imageB);

      //Modify this duplicate.
      modify(duplicateB);

      //Clone the original bitmap to create
another
      // duplicate.
      var duplicateC:Bitmap = new Bitmap(

original.bitmapData.clone());
      //Place the duplicateC below the other two
in the
      // VBox.
      duplicateC.x = 5;
      duplicateC.y = 2*original.height;

      var imageC:Image = new Image();
      imageC.addChild(duplicateC);
      this.addChild(imageC);

      //Modify the pixels as above to add some
color to
      // the image.
      modify(duplicateC);
      //Now invert the colors in the top half of
this
      // bitmap. Note that the magenta and green
colors
      // swap positions.
      invert(duplicateC);

    } //end completeHandler
    //--------------------------------------------------
------//

    //This method modifies the pixels in the
incoming
    // bitmap in a variety of ways.
    private function modify(bitmap:Bitmap):void{
      //Get the BitmapData object from the
incoming
      // Bitmap object.
      var bitmapData:BitmapData =
bitmap.bitmapData;

      //Process pixels using the getPixels and
      // setPixels methods.

      //Get a rectangular array of pixels
comprising
      // 50 columns by 8 rows in a one-dimensional
      // array of bytes. The bytes are ordered in
the
      // array as row 0, row 1, etc.  Each pixel
is
      // represented by four consecutive bytes in
ARGB
      // order.
      var rawBytes:ByteArray = new ByteArray();
      rawBytes = bitmapData.getPixels(
                                new
Rectangle(10,10,50,8));

      //Set the colors of the top four rows to
magenta
      // and the color of the bottom four rows to
      // green. Don't modify alpha.
      var cnt:uint = 1;
      while(cnt < rawBytes.length){
        if(cnt < rawBytes.length/2){
          rawBytes[cnt] = 255;
          rawBytes[cnt + 1] = 0;
          rawBytes[cnt + 2] = 255;
        }else{
          rawBytes[cnt] = 0;
          rawBytes[cnt + 1] = 255;
          rawBytes[cnt + 2] = 0;
        } //end if-else

        cnt += 4;//Increment the counter by 4.
      }//end while loop

      //Put the modified pixels back in the
bitmapData
      // object.
      rawBytes.position = 0;//this is critical
      bitmapData.setPixels(
                      new
Rectangle(10,10,50,8),rawBytes);


      //Process pixels using the setPixel32
method.

      //Put a magenta border on the left edge and
a
      // cyan border on the right edge. Note that
the
```

```
      // byte values in the 32-bit pixel are in
ARGB order
      // and the border thickness is two pixels.
      for(var row:uint = 0;row <
bitmapData.height;

row++){
        bitmapData.setPixel32(0,row,0xFFFF00FF);
        bitmapData.setPixel32(1,row,0xFFFF00FF);
        bitmapData.setPixel32(bitmapData.width -
1,
                              row,0xFF00FFFF);
        bitmapData.setPixel32(bitmapData.width -
2,
                              row,0xFF00FFFF);
      }//end for loop

      //Put a cyan border along the top edge and a
      // magenta border along the bottom edge.
      for(var col:uint = 0;col < bitmapData.width;

col++){
        bitmapData.setPixel32(col,0,0xFF00FFFF);
        bitmapData.setPixel32(col,1,0xFF00FFFF);

bitmapData.setPixel32(col,bitmapData.height - 1,
                              0xFFFF00FF);

bitmapData.setPixel32(col,bitmapData.height - 2,
                              0xFFFF00FF);

      } //End for loop

    } //end modify method
    //--------------------------------------------
------//
```

```
    //This method inverts all of the pixels in the
top
    // half of the incoming bitmap.
    private function invert(bitmap:Bitmap):void{
      //Get the BitmapData object.
      var bitmapData:BitmapData =
bitmap.bitmapData;

      //Get a one-dimensional byte array of pixel
data
      // from the top half of the bitmapData
object
      var rawBytes:ByteArray = new ByteArray();
      rawBytes = bitmapData.getPixels(new
Rectangle(

0,0,bitmapData.width,bitmapData.height/2));

      //Invert the colors by subtracting each
color
      // component value from 255.
      var cnt:uint = 1;
      while(cnt < rawBytes.length){
        rawBytes[cnt] = 255 - rawBytes[cnt];
        rawBytes[cnt + 1] = 255 - rawBytes[cnt +
1];
        rawBytes[cnt + 2] = 255 - rawBytes[cnt +
2];

        cnt += 4;//increment the counter
      }//end while loop

      //Put the modified pixels back in the
bitmapData
      // object.
      rawBytes.position = 0;//this is critical
      bitmapData.setPixels(new Rectangle(

0,0,bitmapData.width,bitmapData.height/2),
                rawBytes);

    } //end invert method
    //--------------------------------------------
------//

  } //end class
} //end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Fundamentals of Image Pixel Processing
- Files:
    - ActionScript0132\ActionScript0132.htm
    - ActionScript0132\Connexions\ActionScriptXhtml0132.htm

-end-

Using Chroma Key Compositing to Create Transparent Backgrounds
Learn how to convert an image with a constant color background into an image with a transparent background. This involves the use of a technique commonly known as chroma key compositing.

Note: Click ChromaKey01 to run this ActionScript program. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

Getting started

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3.

The lesson titled **Using Flex 3 in a Flex 4 World** was added later to accommodate the release of Flash Builder 4. *(See Baldwin's Flex programming website .)*

You should study those lessons before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the

emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

In an earlier lesson titled **Bitmap Basics** , I explained the differences between Flex projects and ActionScript projects. I also introduced you to the classes named **Bitmap** and **BitmapData** .

The fundamentals of pixel processing

In the lesson titled **Fundamentals of Image Pixel Processing** , I showed you how to:

- Load the contents of an image file into a **BitmapData** object encapsulated in a **Bitmap** object.
- Use the **setPixel32** , **getPixels** , and **setPixels** methods to access and modify the color content of the individual pixels that make up an image.

The organization of image information

In that lesson, I also explained how the color and transparency information for an image is stored in files and in the computer's memory. That included topics such as:

- Vector graphics versus bitmap graphics
- Width, height, and color depth
- The RGB and ARGB color models
- The effect of the transparency or alpha value *(total and partial opacity or transparency)*
- Bitmap image file formats *(GIF, JPEG, PNG)*

Creating a transparent background

In this lesson, I will show you how to convert an image with a constant color background *(such as the top image in Figure 1)* into an image with a transparent background *(such as the bottom image in Figure 1)* .

Chroma key compositing

This involves the use of a technique commonly known as *chroma key compositing* .

Here is a little of what Wikipedia has to say on the subject:

> *"Chroma key compositing (or chroma keying) is a technique for compositing two images or frames together in which a color (or a small color range) from one image is removed (or made transparent), revealing another image behind it.*
>
> *This technique is also referred to as color keying, colour-separation overlay (CSO; primarily by the BBC[1]), greenscreen, and bluescreen.*
>
> *It is commonly used for weather forecast broadcasts, wherein the presenter appears to be standing in front of a large map, but in the studio it is actually a large blue or green background. The meteorologist stands in front of a bluescreen, and then different weather maps are added on those parts in the image where the color is blue."*

Commonly used in computer graphics

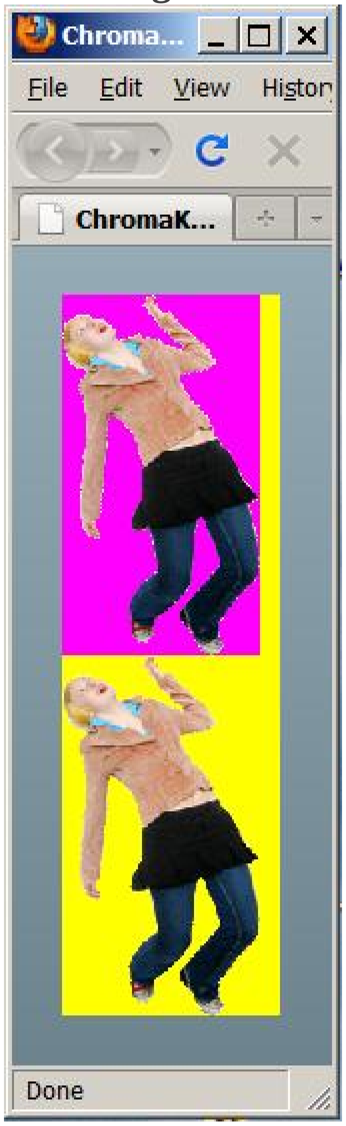
The process is also commonly used in computer graphics where it is desired to overlay one image onto another without letting the background color of the front image show.

That is the intent of this lesson, and the process developed here will be used in a future lesson on animation.

Preview

In this lesson, I will show you to start with an image having a solid background color, such as the top image in Figure 1 and how to convert that image into one with a transparent background, such as the bottom image in Figure 1.
Screen output for the program named ChromaKey01.

Figure



Screen output for the program named ChromaKey01.

A yellow Canvas object

Both images in Figure 1 are displayed on the same yellow **Canvas** object.

The color of the canvas is hidden by the magenta background color of the top image. However, that magenta background is totally transparent in the bottom image, allowing the yellow color of the canvas to show through.

Discussion and sample code

The MXML file

The MXML file, shown in Listing 8, is no different from MXML files used to launch ActionScript programs in earlier lessons. Therefore, no explanation of the MXML code is warranted.

Will explain in fragments

I will explain the ActionScript code for the program named **ChromaKey01** in fragments. A complete listing of the code for the ActionScript class named **Driver** is provided in Listing 7.

The processChromaKey method

This program illustrates the use of a custom method named **processChromaKey** . This method scans all of the pixels in an incoming bitmap and identifies those pixels with a color of pure magenta. The alpha values for the magenta pixels are set to zero to make them transparent. Then the bitmap is modified accordingly.

The method can easily be modified to accommodate an image background of any solid color such as green or blue. Note however, that magenta is commonly used in computer graphics, and is often referred to as *"magic pink."*

Beginning of the ActionScript class named Driver

The Flex code in Listing 8 instantiates a new object of the class named **Driver** and adds that object to the display list as a child of the **Application** container.

The ActionScript code for the class named **Driver** begins in Listing 1.

Example:
Beginning of the class named Driver.

```
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.geom.Rectangle;
  import flash.utils.ByteArray;
  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.events.FlexEvent;

//==================================================
====//

  public class Driver extends Canvas {
    private var origImage:Image = new Image();
```

Extend the Flex Canvas class

Note that the **Driver** class extends the Flex **Canvas** class, thus making it possible to add other objects as children of an object of the **Driver** class.

Another reason for extending the Canvas class

I could have accomplished that purpose by extending any number of Flex container classes. However, an important characteristic of the **Canvas** class for this application is the ability to specify the physical locations of the child objects in absolute coordinates. Few if any other Flex containers allow you to do that,

The constructor

The constructor for the **Driver** class is shown in its entirety in Listing 2.

Example:
The constructor for the class named Driver.

```
    public function Driver(){//constructor
      //Make this Canvas visible with a yellow 
background.
      setStyle("backgroundColor",0xFFFF00);
      setStyle("backgroundAlpha",1.0);

      //Load the file mage and embed it in the swf 
file.
      //Note the slash that is required by 
FlashDevelop.
      [Embed("/dancer.png")]
      var img:Class;
      origImage.load(img);

      //Register a CREATION_COMPLETE listener

this.addEventListener(FlexEvent.CREATION_COMPLETE,

completeHandler);
    } //end constructor
```

Make the canvas visible

The constructor begins by setting two style properties on the **Canvas** object to make it yellow and opaque.

You learned how to set styles on objects in ActionScript code as early as the lesson titled **What is OOP and Why Should I Care?**

Load the image from the file

Then the constructor loads the image from the image file named **dancer.png** and embeds it in the swf file. You learned how to do this as early as the lesson titled **Inheritance - The Big Picture.**

No lossy compression allowed

Note that this program uses an image from a ***png*** file for demonstration purposes.

Unlike a ***jpg*** file, *(which uses lossy compression)* the colors that you extract from a png file are exactly the colors that were stored in the png file. That is a requirement for a chroma key process that is based on an exact color match.

An unusual requirement

There is an unusual requirement shown in Listing 2 that you haven't seen in the earlier lessons in this series.

This is the first lesson that I have written explaining a program that loaded and embedded an image file where the program was developed using **FlashDevelop** *(see Publication Issues below).*

File organization

In this case, the image file was placed in the same folder as the MXML file. Note that it was necessary for me to precede the name of the image file in Listing 2 with a slash character. I have not previously encountered that requirement when using Flex Builder 3. *(Note however that inclusion of the slash character in a Flex Builder 3 project doesn't seem to cause a problem.)*

Register a CREATION_COMPLETE listener

Finally, Listing 2 registers an event handler to handle CREATION_COMPLETE events fired by the **Canvas** object.

You learned how and why to use a CREATION_COMPLETE listener in the earlier lesson titled **Encapsulation - The Big Picture.**

The CREATION_COMPLETE event handler

The CREATION_COMPLETE event handler is shown in its entirety in Listing 3. This method is executed when the **Canvas** object has been fully created.

Example:
The CREATION_COMPLETE event handler.

```
    private function completeHandler(

event:mx.events.FlexEvent):void{
      //Get and save a reference to a Bitmap
object
      // containing the contents of the origImage
file.
      var origBitMap:Bitmap =
Bitmap(origImage.content);

      //Set the width and height of the Canvas
object
      // based on the size of the origBitMap
bitmap. Make
      // ten pixels wider and twice as high as the
      // bitmap.
      this.width = origBitMap.width + 10;
      this.height = 2*origBitMap.height;

      //Add the original image to the Canvas
object at
      // the default location of 0,0.
      this.addChild(origImage);
      //Clone the origBitMap to create a
      // duplicate.
      var dupBitMap:Bitmap = new Bitmap(

origBitMap.bitmapData.clone());

      //Put the dupBitMap in a new Image object
and
      // place it on the canvas below the original
image.
      dupBitMap.x = 0;
      dupBitMap.y = origBitMap.height;

      var newImage:Image = new Image();
      newImage.addChild(dupBitMap);
      this.addChild(newImage);

      //Set the alpha value for all pixels in the
new
      // image with a color of pure magenta to
zero.
      processChromaKey(dupBitMap);

    } //end completeHandler
```

The call to the method named processChromaKey

With the exception of the call to the method named **processChromaKey** at the end of Listing 3, I explained everything in Listing 3 in the earlier lesson titled **Fundamentals of Image Pixel Processing.** I won't waste your time by repeating that explanation here.

Beginning of the *processChromaKey* method

The **processChromaKey** method begins in Listing 4.

Example:
Beginning of the processChromaKey method.

```
    private function
processChromaKey(bitmap:Bitmap):void{
      //Get the BitmapData object.
      var bitmapData:BitmapData =
bitmap.bitmapData;

      //Populate a one-dimensional byte array of
pixel
      // data from the bitmapData object. Note
that the
      // pixel data format is ARGB.
      var rawBytes:ByteArray = new ByteArray();
      rawBytes = bitmapData.getPixels(new
Rectangle(

0,0,bitmapData.width,bitmapData.height));
```

A reference to a Bitmap object

This method receives a reference to a **Bitmap** object as an incoming parameter.

It identifies all of the pixels in the incoming bitmap with a pure magenta color and sets the alpha bytes for those pixels to a value of zero.

This causes those pixels to become transparent as shown by the bottom image in Figure 1.

Nothing new in Listing 4

I explained all of the code in Listing 4 in the earlier lesson titled **Fundamentals of Image Pixel Processing.**

The ByteArray object is populated with pixel data

When the **getPixels** method returns in Listing 4, the pixels from a rectangular region that encompasses the entire bitmap are stored in the **ByteArray** object referred to by **rawBytes** .

The organization of the pixel data

The array is populated with the bitmap pixel data from the rectangular region on a row by row basis.

The first four bytes in the array belong to the pixel in the upper-left corner of the rectangular region. The next four bytes belong to the pixel immediately to the right of that one, and so on.

Four bytes per pixel

Each set of four bytes represents one pixel in ARGB format. In other words, the first byte in the four-byte group is the alpha byte. That byte is followed by the red byte, the green byte, and the blue byte in that order.

This information is critical when time comes to use the data in the array to modify the bitmap data.

Set selected alpha values to zero

The code in Listing 5 iterates through the entire set of image pixels and sets selected alpha values to zero.

Example:
Set selected alpha values to zero.

```
      //Declare working variables. Note that there
is
      // no requirement to deal with the green
color
      // value in this case of testing for
magenta.
      var cnt:uint;
      var red:uint;
      var green:uint;
      var blue:uint;

      for (cnt = 0; cnt < rawBytes.length; cnt +=
4) {
        //alpha is in rawBytes[cnt]
        red = rawBytes[cnt + 1];
        green = rawBytes[cnt + 2];
        blue = rawBytes[cnt + 3];

        if ((red == 255) && (green == 0 ) &&
                                          (blue ==
255)) {
          //The color is pure magenta. Set the
value
          // of the alpha byte to zero.
          rawBytes[cnt] = 0;
        }//end if
      }//end for loop
```

A for loop

After declaring some working variables, Listing 5 iterates through all of the data in the **ByteArray** object. It extracts the red, green, and blue color bytes from each four-byte group and tests to see if red and blue are both set to full intensity with a value of 255 and green is set to zero.

If true, this is interpreted to match the magenta background color, and the value of the alpha byte in that four-bit group is set to zero.

Store the modified pixel data in the bitmap

Listing 6 copies the modified pixel data from the **ByteArray** object back into the bitmap, overwriting the pixel data previously stored in the bitmap.

Example:
Store the modified pixel data in the bitmap.

```
      //Put the modified pixels back into the 
bitmapData
      // object.
      rawBytes.position = 0;//this is critical
      bitmapData.setPixels(new Rectangle(

0,0,bitmapData.width,bitmapData.height),
                rawBytes);

    } //end processChromaKey method
    //--------------------------------------------
------//

  } //end class
} //end package
```

I explained all of the code in Listing 6 in the earlier lesson titled **Fundamentals of Image Pixel Processing.**

The end of the class

Listing 6 also signals the end of the method, the end of the class, the end of the package, and the end of the program.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 7 and Listing 8. Use that code to create a new project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings for the program discussed in this lesson are provided in Listing 7 and Listing 8 below.

Example:
Listing of the class named Driver.

```
/*Project ChromaKey01
This project scans all of the pixels in an image
to
identify those pixels with a color of pure
magenta. The
alpha value for those pixels is set to zero to

make them
transparent.
*************************************************
```

```
*******/
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.geom.Rectangle;
  import flash.utils.ByteArray;
  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.events.FlexEvent;

//==================================================
====//

  public class Driver extends Canvas {
    private var origImage:Image = new Image();

    public function Driver(){//constructor
      //Make this Canvas visible with a yellow
background.
      setStyle("backgroundColor",0xFFFF00);
      setStyle("backgroundAlpha",1.0);

      //Load the origImage and embed it in the swf
file.
      //Note the slash that is required by
FlashDevelop.
      [Embed("/dancer.png")]
      var img:Class;
      origImage.load(img);

      //Register a CREATION_COMPLETE listener

this.addEventListener(FlexEvent.CREATION_COMPLETE,


completeHandler);
    } //end constructor
    //------------------------------------------
```

```
------//

    //This handler method is executed when the
Canvas has
    // been fully created.
    private function completeHandler(

event:mx.events.FlexEvent):void{
      //Get and save a reference to a Bitmap
object
      // containing the contents of the origImage
file.
      var origBitMap:Bitmap =
Bitmap(origImage.content);

      //Set the width and height of the Canvas
object
      // based on the size of the origBitMap
bitmap. Make
      // ten pixels wider and twice as high as the
      // bitmap.
      this.width = origBitMap.width + 10;
      this.height = 2*origBitMap.height;

      //Add the original image to the Canvas
object at
      // the default location of 0,0.
      this.addChild(origImage);

      //Clone the origBitMap to create a
      // duplicate.
      var dupBitMap:Bitmap = new Bitmap(

origBitMap.bitmapData.clone());


      //Put the dupBitMap in a new Image object
and
```

```
      // place it on the canvas below the original
image.
      dupBitMap.x = 0;
      dupBitMap.y = origBitMap.height;

      var newImage:Image = new Image();
      newImage.addChild(dupBitMap);
      this.addChild(newImage);

      //Set the alpha value for all pixels in the
new
      // image with a color of pure magenta to
zero.
      processChromaKey(dupBitMap);

    } //end completeHandler
    //-------------------------------------------------
------//

    //This method identifies all of the pixels in
the
    // incoming bitmap with a pure magenta color
and sets
    // the alpha bytes for those pixels to a value
of
    // zero.
    private function
processChromaKey(bitmap:Bitmap):void{
      //Get the BitmapData object.
      var bitmapData:BitmapData =
bitmap.bitmapData;

      //Populate a one-dimensional byte array of
pixel
      // data from the bitmapData object. Note
that the
      // pixel data format is ARGB.
      var rawBytes:ByteArray = new ByteArray();
      rawBytes = bitmapData.getPixels(new
Rectangle(

0,0,bitmapData.width,bitmapData.height));

      //Declare working variables. Note that there
is
      // no requirement to deal with the green
color
      // value in this case of testing for
magenta.
      var cnt:uint;
      var red:uint;
      var green:uint;
      var blue:uint;

      for (cnt = 0; cnt < rawBytes.length; cnt +=
4) {
        //alpha is in rawBytes[cnt]
        red = rawBytes[cnt + 1];
        green = rawBytes[cnt + 2];
        blue = rawBytes[cnt + 3];

        if ((red == 255) && (green == 0 ) &&
                                          (blue ==
255)) {
          //The color is pure magenta. Set the
value
          // of the alpha byte to zero.
          rawBytes[cnt] = 0;
        }//end if

      }//end for loop


      //Put the modified pixels back into the
bitmapData
```

```
      // object.
      rawBytes.position = 0;//this is critical
      bitmapData.setPixels(new Rectangle(

0,0,bitmapData.width,bitmapData.height),
                rawBytes);

    } //end processChromaKey method
    //--------------------------------------------------
------//

  } //end class
} //end package
```

Example:
Listing of the MXML file.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
Project ChromaKey01
This project scans all of the pixels in an image
to
identify those pixels with a color of pure
magenta. The
alpha value for those pixels is set to zero to
make them
transparent.
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>
```

```
</mx:Application>
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Using Chroma Key Compositing to Create Transparent Backgrounds
- Files:
  - ActionScript0134\ActionScript0134.htm
  - ActionScript0134\Connexions\ActionScriptXhtml0134.htm

Note: PDF disclaimer: Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

Note: Publication issues : This project was originally developed using the free open-source FlashDevelop development tool. However, because of limitations of the Connexions web site, it was necessary for me to convert the project to a Flex Builder 3 project to make it possible for you to run the program from this Connexions module.

-end-

Drag and Drop Basics
Learn the basics of writing ActionScript 3 code to provide a drag and drop capability.

Note: Click DragAndDrop01 to run this ActionScript program. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming (OOP) with ActionScript.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General Background Information

For Adobe online documentation on this topic, see Using Drag and Drop .

A drag and drop operation is carried out in three stages:

- Initiation
- Dragging
- Dropping

Initiation

As you are probably already aware, initiation consists of the user pointing to an item with the mouse and pressing the mouse button.

Dragging

During the drag operation, the user drags the item to another location on the screen without releasing the mouse button.

Dropping

When the item has been dragged to the new location, the user releases the mouse button causing the item to remain in the new location.

Copying

It is also possible to copy an item using the drag and drop gestures, but that capability won't be illustrated in this lesson. Instead, this lesson will concentrate on moving a Flex component from one location in its container to a different location in its container.

Classes and events

The sample program that I will explain in this lesson will use the following classes, methods, and events:

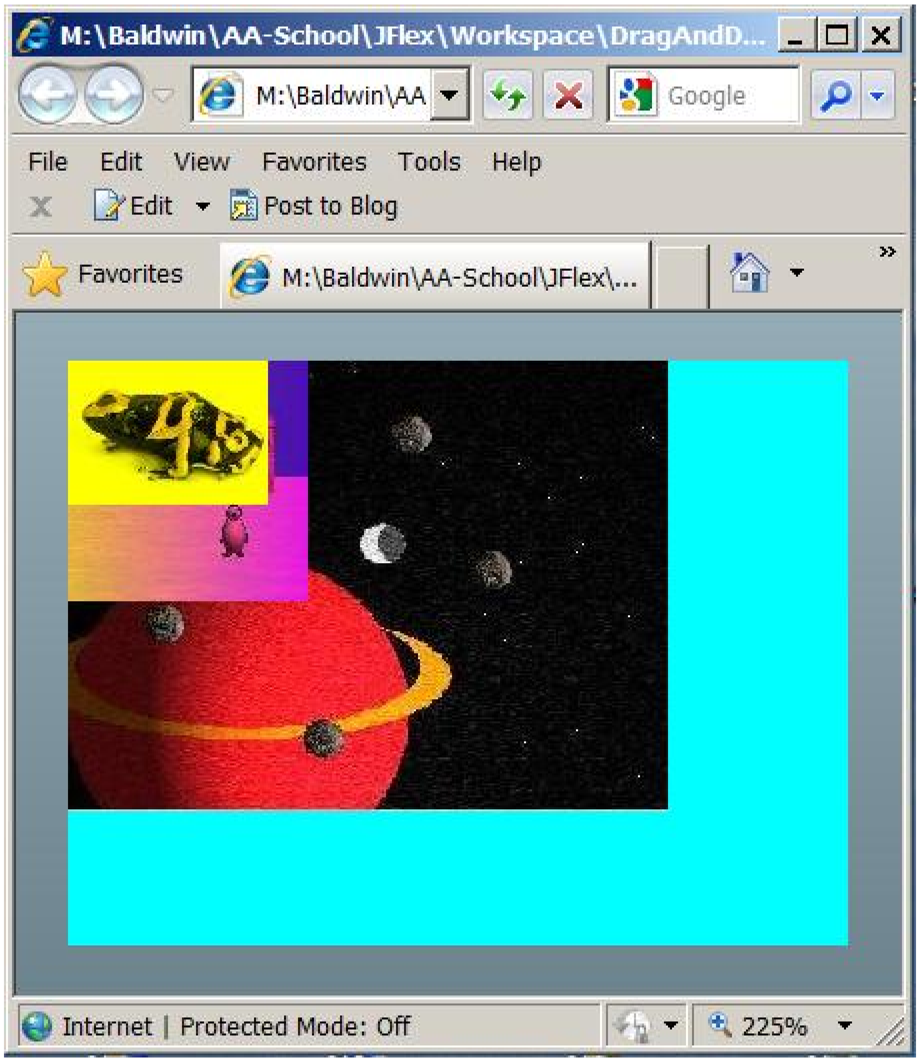
- MouseEvent class
  - mouseDown event
- DragEvent class
  - dragDrop event
  - dragEnter event
- DragManager class
  - acceptDragDrop method
  - doDrag method
- DragSource class
  - addData method
  - hasFormat method

Preview

I will explain a program named **DragAndDrop01** . This program illustrates the fundamentals of drag and drop in ActionScript 3. The program places three images in the upper-left corner of a **Canvas** object as shown in Figure 1.

Program output at startup.

Figure

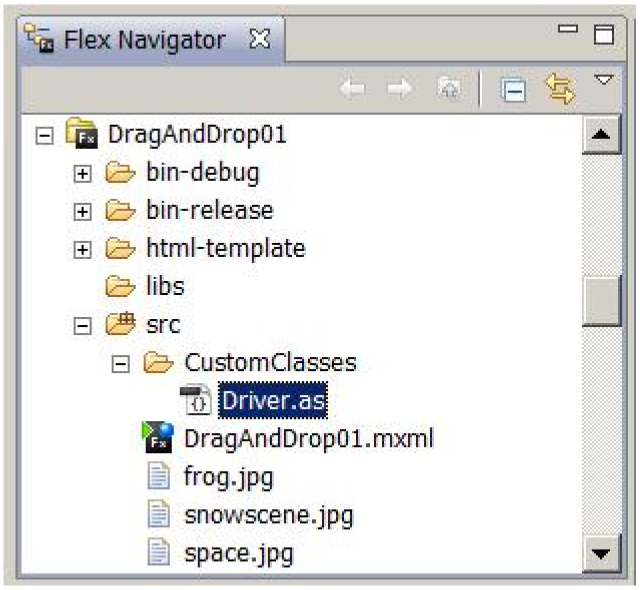

M:\Baldwin\AA-School\JFlex\Workspace\DragAndD...
M:\Baldwin\AA
Google
File Edit View Favorites Tools Help
Edit
Post to Blog
Favorites
M:\Baldwin\AA-School\JFlex\...
Internet | Protected Mode: Off
225%

Program output at startup.

The program file structure

The program file structure, taken from the Flex Builder 3 Navigator panel is as shown in Figure 2.
Program file structure.

Figure



Program file structure.

Three image files

As you can see in Figure 2, the program uses **the following image files** :

- 0 - space.jpg
- 1 - snowscene.jpg
- 2 - frog.jpg

The z-axes indices

The program sets the z-axis indices in the order shown in the above list on the **Image** objects produced using the image files. This causes the *space* image to be in the back (0), the *frog* image to be in the front (2), and the *snowscene* image to be in the middle (1).

The three images are initially placed in the upper-left corner of the canvas, which is shown as a cyan rectangle in Figure 1.

Any image can be dragged

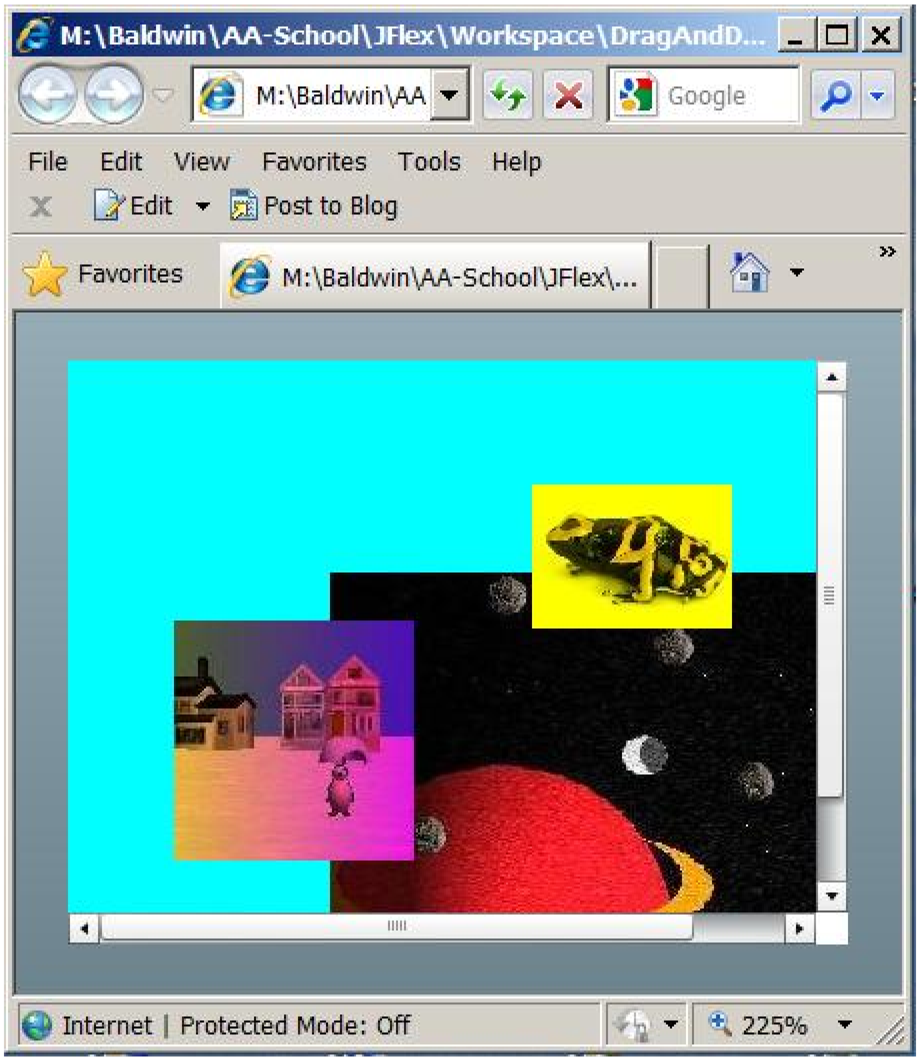
If you run this program, you will see that any any of the images can be dragged and dropped anywhere within the canvas as long as the mouse pointer doesn't leave the canvas. However, if the edge of the dragged image goes outside the left edge or the top of the canvas, the drag and drop operation is aborted.

If the dragged image goes outside the right side or the bottom of the canvas, scroll bars automatically appear on the canvas as shown in Figure 3.

Program output after dragging the images

Figure 3 shows the program output after dragging the three images to different locations.
Program output after dragging the images.

Figure


M:\Baldwin\AA-School\JFlex\Workspace\DragAndD...
M:\Baldwin\AA
Google
File Edit View Favorites Tools Help
Edit
Post to Blog
Favorites
M:\Baldwin\AA-School\JFlex\...
Internet | Protected Mode: Off
225%

Program output after dragging the images.

Discussion and sample code

Will explain in fragments

I will explain the code for this program in fragments. Complete listings of the MXML code and the ActionScript code are provided in Listing 16 and Listing 17 near the end of the lesson.

The MXML file

The MXML file is shown in Listing 1 and also in Listing 16 for your convenience.

Example:
he MXML file for DragAndDrop01.

```
<?xml version="1.0" encoding="utf-8"?>
<!--DragAndDrop01-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

As you can see, the MXML file is very simple because the program was coded almost entirely in ActionScript. The MXML code simply instantiates an object of the **Driver** class. From that point forward, the behavior of the program is controlled by ActionScript code.

The ActionScript file

Beginning of the Driver class

The Driver class begins in Listing 2.

Example:
Beginning of the Driver class for DragAndDrop01.

```
package CustomClasses{
  import flash.events.MouseEvent;

  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.core.DragSource;
  import mx.events.DragEvent;
  import mx.events.FlexEvent;
  import mx.managers.DragManager;


//==================================================
====//

  public class Driver extends Canvas {
    private var imageA:Image = new Image();
    private var imageB:Image = new Image();
    private var imageC:Image = new Image();
    private var localX:Number;
    private var localY:Number;
```

Extends the Canvas class

As you can see in Listing 2, the **Driver** class extends the **Canvas** class. Therefore, an object of the **Driver** class is a **Canvas** object and has all of the attributes associated with a **Canvas** object. Among those attributes is the following, which was taken from the documentation :

> "A Canvas layout container defines a rectangular region in which you place child containers and controls. It is the only container

> that lets you explicitly specify the location of its children within the container by using the x and y properties of each child."

As you will see, the new location of each image is explicitly specified each time it is dragged to a new location.

Instantiate three new Image objects

The code in Listing 2 instantiates three new **Image** objects, which will be loaded with the contents of the three image files listed earlier . The code in Listing 2 also declares two instance variables that will be used to store the position of the mouse pointer within an image when the drag operation is initiated.

Beginning of the constructor for the Driver class

The constructor for the **Driver** class begins in Listing 3.

Example:
Beginning of the constructor for the Driver class.

```
    public function Driver(){//constructor
      setStyle("backgroundColor",0x00FFFF);
      setStyle("backgroundAlpha",1.0);
```

Make the canvas visible

Normally a **Canvas** object is not visible. The code in Listing 3 sets the alpha value for the **Canvas** object to 1.0 making it opaque and visible. Listing 3 also sets the background color of the **Canvas** object to cyan as shown in Figure 1.

The size of the **Canvas** object will be set later when the **Canvas** object and all of its children have been constructed.

Prepare the three images

Listing 4 prepares the three images for use by the program.

Example:
Prepare the three images.

```
      //Embed the image files in the SWF file.
      [Embed("snowscene.jpg")]
      var imgA:Class;

      [Embed("space.jpg")]
      var imgB:Class;

      [Embed("frog.jpg")]
      var imgC:Class;

      //Load the images from the embedded image
files
      // into the Image objects.
      imageA.load(imgA);
      imageB.load(imgB);
      imageC.load(imgC);

      // Set the z-axes indices such that the frog
is
      // in front, the snowscene is in the middle
and the
      // space image is at the back.
      addChildAt(imageB,0);//set index to 0
      addChildAt(imageA,1);//set index to 1
      addChildAt(imageC,2);//set index to 2
```

Listing 4 begins by embedding the three image files in the SWF file. Then it loads the contents of the image files into the **Image** objects instantiated in Listing 2. Finally Listing 3 adds the **Image** objects as children of the **Canvas** object.

Set the z-axis indices

The z-axis index of each **Image** object is set in Figure 4 so as to place the space image at the back, the frog image at the front, and the snowscene image between the other two.

Register a creationComplete event handler

Listing 5 registers a **creationComplete** event handler on the **Canvas** object. This event handler will be executed after the **Canvas** object and all of its children are fully constructed.

Example:
Register a creationComplete event handler.

```
this.addEventListener(FlexEvent.CREATION_COMPLETE,

completeHandler);
    } //end constructor
```

Beginning of the creationComplete event handler

The **creationComplete** event handler begins in Listing 6. This handler is executed once when the **Canvas** object and all of its children have been constructed.

Example:
Beginning of the creationComplete event handler.

```
    private function completeHandler(

event:mx.events.FlexEvent):void{
      //Set the width and height of the canvas
based on
      // the dimensions of imageB.
      this.width = 1.3*imageB.width;
      this.height = 1.3*imageB.height;
```

Set the size of the Canvas object

Listing 6 sets the width and height of the **Canvas** object based on the dimensions of the **Image** object referred to by **imageB** . It was not possible to reliably execute this code in the constructor because the code might be executed before the contents of the image file were fully loaded into the **Image** object.

Register a mouseDown event handler on each Image object

A drag and drop operation is heavily dependent on the handling of different types of events. The remaining code in the **creationComplete** event handler registers appropriate event handlers on the images and on the **Canvas** object to support the drag and drop operation with the **Canvas** object as the drag target.

As you will see later, a drag operation is initialized when an image dispatches a **mouseDown** event. Listing 7 registers the same **mouseDown** event handler on all three **Image** objects.

Example:

Register a mouseDown event handler on each Image object.

```
imageA.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);

imageB.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);

imageC.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);
```

Register dragDrop and dragEnter event handlers on the Canvas object

Two different event handlers must be registered on the drag target, which is the **Canvas** object in this case. The registration of those event handlers on the **Canvas** object is accomplished in Listing 8.

Example:
Register dragDrop and dragEnter event handlers on the Canvas object.

```
      this.addEventListener(DragEvent.DRAG_DROP,

dropHandler);
      this.addEventListener(DragEvent.DRAG_ENTER,

enterHandler);
    } //end completeHandler
```

Beginning of the mouseDown event handler

The **mouseDown** event handler that was registered on the **Image** objects in Listing 7 begins in Listing 9. This event handler initiates the drag and drop operation on the **Image** object that dispatches the event.

Example:
Beginning of the mouseDown event handler.

```
    private function mouseDownHandler(

event:MouseEvent):void{

      //Save the location of the mouse within the
image
      // being dragged. This information will be
used
      // later to properly position the dropped
image in
      // the drop target.
      this.localX = event.localX;
      this.localY = event.localY;
```

Positioning the dropped object

The easiest approach simply drops the image with its upper-left corner at the position of the mouse pointer when the mouse button is released. However, in my opinion, that is somewhat less than satisfactory from a visual viewpoint.

The drag proxy

When you drag an image, there is a default drag proxy that moves along with the mouse. *(It is possible to replace the default drag proxy with a drag proxy of your choice.)* The default drag proxy is a partially transparent rectangle that is the same size as the image.

Adjust the position of the upper-left corner

My preference is to manually adjust the drop location of the image based on the upper-left corner of the drag proxy and not based on the location of the mouse pointer. The code in Listing 9 gets and saves the coordinates of the mouse pointer within the image when the event is dispatched. As you will see later, I use these coordinates later to set the drop location on the basis of the upper-left corner of the proxy.

Get and save the drag initiator

The documentation refers to the object being dragged as the *drag initiator* .

Example:
Get and save the drag initiator.

```
      //Get the drag initiator component from the 
event
      // object and cast it to the correct type.
      var dragInitiator:Image = Image(

event.currentTarget);
```

In this program, the drag initiator could be any of the three images shown in Figure 1 and Figure 3.

The code in Listing 10

- Gets a reference to the **Image** object that dispatched the **mouseDown** event from the incoming method parameter
- Casts it to type **Image** , and
- Saves it in the variable named **dragInitiator** .

Populate a DragSource object with a copy of the image being dragged

Here is part of what the documentation has to say about the **DragSource** class.

> "The DragSource class contains the data being dragged. The data can be in multiple formats, depending on the type of control that initiated the drag.
>
> Each format of data is identified with a string. ... Data can be added directly using the addData() method, or indirectly using the addHandler() method."

Listing 11 adds the image being dragged to a new **DragSource** object and provides an identifier for the format as a string. You will see later how this string is used to establish the drop target.

Example:
Populate a DragSource object with a copy of the image being dragged.

```
      var dragSource:DragSource = new
DragSource();


dragSource.addData(dragInitiator,"imageObject");
```

Initiate the drag and drop operation by calling the doDrag method

Listing 12 initiates the drag and drop operation by calling the static **doDrag** method of the **DragManager** class.

Example:
Initiate the drag and drop operation by calling the doDrag method.

```
DragManager.doDrag(dragInitiator,dragSource,event)
;

    }//end mouseDownHandler
```

What does the documentation have to say about the DragManager class?

Here is part of what the documentation has to say about the **DragManager** class.

> "The **DragManager** class manages drag and drop operations, which let you move data from one place to another in a Flex application. For example, you can select an object, such as an item in a List control or a Flex control, such as an Image control, and then drag it over another component to add it to that component.
>
> All methods and properties of the DragManager are static, so you do not need to create an instance of it. ...
>
> When the user selects an item with the mouse, the selected component is called the drag initiator. The image displayed

> during the drag operation is called the drag proxy.
>
> When the user moves the drag proxy over another component, the **dragEnter** event is sent to that component. If the component accepts the drag, it becomes the drop target and receives dragOver, dragExit, and dragDrop events.
>
> When the drag is complete, a dragComplete event is sent to the drag initiator."

What about the doDrag method?

The documentation states simply that the **doDrag** method

"Initiates a drag and drop operation."

The doDrag method parameters

The **doDrag** method has several parameters with default values in addition to the three shown in Listing 12. Here is part of what the documentation has to say about the three parameters passed to the **doDrag** method in Listing 12.

- **dragInitiator :IUIComponent** - IUIComponent that specifies the component initiating the drag.
- **dragSource :DragSource** - DragSource object that contains the data being dragged.
- **event:MouseEvent** - The MouseEvent that contains the mouse information from the start of the drag.

The end of the mouseDown event handler

Listing 12 signals the end of the **mouseDown** event handler. This leaves two more event handlers to be discussed. The remaining two event handlers were registered on the drop target *(**Canvas** object)* by the code in Listing 8.

The dragEnter event handler

As you learned earlier , when the user moves the drag proxy over another component, that component dispatches a **dragEnter** event.

If a **dragEnter** event handler has been registered on that component, the handler method is executed. If the code in the event handler accepts the drag, it becomes the drop target and receives **dragOver** , **dragExit** , and **dragDrop** events.

In this case the intended drop target is the **Canvas** object and the event handler shown in Listing 13 is registered on that object.

Example:
The dragEnter event handler.

```
    private function
enterHandler(event:DragEvent):void{
      if
(event.dragSource.hasFormat("imageObject")){
          DragManager.acceptDragDrop(

Canvas(event.currentTarget));
      } //end if
    } //end enterHandler
```

Confirm the correct format string

The code in Listing 13 checks to confirm that the format string in the **DragSource** object matches "imageObject" (see Listing 11). If so, it calls the static **acceptDragDrop** method on the **DragManager** class, passing a reference to itself as a parameter in the method call.

Accept the dragged object

The call to the **acceptDragDrop** method notifies the **DragManager** that the **Canvas** object is willing to accept the contents of the **DragSource** object being dropped onto itself.

Beginning of the dragDrop event handler

The **dragDrop** event handler was registered on the **Canvas** object in Listing 8. This method is executed after the **Canvas** object accepts the drag and the user releases the mouse button while the drag proxy is over the **Canvas** .

Correct the drop position for the image

The code in Listing 14 uses the current location of the mouse pointer along with the values stored in **localX** and **localY** to compute the new location for the upper-left corner of the image when it is dropped on the canvas.

> *(Recall that **localX** and **localY** contain the coordinates of the mouse pointer relative to the upper-left corner of the image when the **mouseDown** event was dispatched by the image and the drag and drop operation began.)*

I explained the need for this position adjustment earlier .

Example:
Beginning of the dragDrop event handler.

```
    private function
dropHandler(event:DragEvent):void{

      var cornerX:Number =
(Canvas(event.currentTarget).
                                         mouseX) -
```

```
localX;
      var cornerY:Number =
(Canvas(event.currentTarget).
                                         mouseY) -
localY;
```

Do the drop

Listing 15 checks to confirm that the location at which the upper-left corner of the image will be placed is within the bounds of the canvas on the left side and the top. If so, it sets the coordinates of the **Image** object that dispatched the original **mouseDown** event to the coordinates that were computed in Listing 14. This causes that **Image** object to move to the new position on the **Canvas** object.

Example:
Do the drop.

```
      if((cornerX > 0.0) && (cornerY > 0.0)){
        Image(event.dragInitiator).x = cornerX;
        Image(event.dragInitiator).y = cornerY
      } //end if
    } //end dropHandler
    //--------------------------------------------
------//

  } //end class
} //end package
```

The end of the program

Listing 15 also signals the end of the **dragDrop** event handler, the end of the **Driver** class, and the end of the program.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 16 and Listing 17. Use that code to create a Flex project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings for the MXML and ActionScript code discussed in this lesson are provided in Listing 16 and Listing 17 below.

Example:
The MXML file for DragAndDrop01.

```
<?xml version="1.0" encoding="utf-8"?>
<!--DragAndDrop01
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>
```

```
</mx:Application>
```

Example:
The ActionScript file for DragAndDrop01.

```
/*DragAndDrop01
Illustrates the fundamentals of drag and drop in
ActionScript 3.

Places three images in a Canvas object:

0 - space.jpg - largest
1 - snowscene.jpg - midsize
2 - frog.jpg - smallest

Sets the z-axis indices as shown above. This
causes the
space image to be in the back, the frog image to
be in the
front, and the snowscene image to be in the
middle.

Any of the images can be dragged and dropped
anywhere
within the canvas so long as the mouse pointer
doesn't
leave the canvas. However, If the edge of the
dragged
image goes outside the left edge or the top of the
canvas,

the drag and drop operation is aborted. If the
dragged
image goes outside the right side or the bottom of
```

```
the
canvas, scroll bars automatically appear on the
canvas.

The size of the canvas is based on the size of the
space
image so that other images can be substituted for
mine
when the program is recompiled so long as the file
names
and paths are the same.
**************************************************
*******/
package CustomClasses{
  import flash.events.MouseEvent;

  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.core.DragSource;
  import mx.events.DragEvent;
  import mx.events.FlexEvent;
  import mx.managers.DragManager;


//==================================================
====//

  public class Driver extends Canvas {
    private var imageA:Image = new Image();
    private var imageB:Image = new Image();
    private var imageC:Image = new Image();
    private var localX:Number;
    private var localY:Number;


    public function Driver(){//constructor
      //Make the Canvas visible.
      setStyle("backgroundColor",0x00FFFF);
      setStyle("backgroundAlpha",1.0);

      //Embed the image files in the SWF file.
      [Embed("snowscene.jpg")]
      var imgA:Class;

      [Embed("space.jpg")]
      var imgB:Class;

      [Embed("frog.jpg")]
      var imgC:Class;

      //Load the images from the embedded image
files
      // into the Image objects.
      imageA.load(imgA);
      imageB.load(imgB);
      imageC.load(imgC);

      // Set the z-axes indices such that the frog
is
      // in front, the snowscene is in the middle
and the
      // space image is at the back.
      addChildAt(imageB,0);//set index to 0
      addChildAt(imageA,1);//set index to 1
      addChildAt(imageC,2);//set index to 2

      //Register an event handler that will be
executed
      // whcn the canvas and its children are
fully
      // constructed.


this.addEventListener(FlexEvent.CREATION_COMPLETE,

completeHandler);
```

```
    } //end constructor
    //--------------------------------------------------
------//

    //This handler method is executed when the
Canvas and
    // its children have been fully created.
    private function completeHandler(

event:mx.events.FlexEvent):void{
      //Set the width and height of the canvas
based on
      // the dimensions of imageB.
      this.width = 1.3*imageB.width;
      this.height = 1.3*imageB.height;

      //Register event listeners to support drag
and drop
      // operations on all three images with the
canvas
      // as the drag target.

imageA.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);

imageB.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);

imageC.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);


      this.addEventListener(DragEvent.DRAG_DROP,

dropHandler);
```

```
      this.addEventListener(DragEvent.DRAG_ENTER,

enterHandler);
    } //end completeHandler
    //--------------------------------------------------
------//

    // This event handler initiates the drag-and-
drop \
    // operation for the image that dispatches the
    // mouseDown event.
    private function mouseDownHandler(

event:MouseEvent):void{

      //Save the location of the mouse within the
image
      // being dragged. This information will be
used
      // later to properly position the dropped
image in
      // the drop target.
      this.localX = event.localX;
      this.localY = event.localY;

      //Get the drag initiator component from the
event
      // object and cast it to the correct type.
      var dragInitiator:Image = Image(

event.currentTarget);

      //Add the image being dragged to a
DragSource
      // object and define an identifier as a
string.
      var dragSource:DragSource = new
```

```
DragSource();

dragSource.addData(dragInitiator,"imageObject");

      //Call the static doDrag method on the
DragManager
      // class to manage the overall drag and drop
      // operation.

DragManager.doDrag(dragInitiator,dragSource,event)
;
    }//end mouseDownHandler
    //--------------------------------------------
------//

    //This dragEnter event handler causes the
canvas to
    // be a suitable drop target.
    private function
enterHandler(event:DragEvent):void{
      if
(event.dragSource.hasFormat("imageObject")){
          DragManager.acceptDragDrop(

Canvas(event.currentTarget));
      } //end if
    } //end enterHandler
    //--------------------------------------------
------//

    //Execute the dragDrop event handler to drop
the image
    // in its new location. Compensate for the
fact that

    // the mouse pointer is not at the upper-left
corner
    // of the image. Also don't allow the image to
```

```
be
    // dragged off the left side of the canvas or
off the
    // top of the canvas.
    private function
dropHandler(event:DragEvent):void{

      //Compute the position of the upper-left
corner of
      // the dropped image.
      var cornerX:Number =
(Canvas(event.currentTarget).
                                         mouseX) -
localX;
      var cornerY:Number =
(Canvas(event.currentTarget).
                                         mouseY) -
localY;
      if((cornerX > 0.0) && (cornerY > 0.0)){
        Image(event.dragInitiator).x = cornerX;
        Image(event.dragInitiator).y = cornerY
      } //end if
    } //end dropHandler
    //--------------------------------------------------
------//

  } //end class
} //end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Drag and Drop Basics
- Files:
    - ActionScript0140\ActionScript0140.htm
    - ActionScript0140\Connexions\ActionScriptXhtml0140.htm

-end-

Dragging Objects between Containers
Learn how to drag objects between containers while giving each container the ability to either accept or reject the drop based on the type of object being dropped.

Note: Click DragAndDrop04 to run this ActionScript program. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming (OOP) with ActionScript.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Most of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. *(See Baldwin's Flex programming website .)* You should study that lesson before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

- Listing 14 . The ActionScript code for the program named DragAndDrop04.

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

In the earlier lesson titled **Drag and Drop Basics** , you learned the basics of writing ActionScript 3 code to provide a drag and drop capability. In this lesson, you will expand on that knowledge by learning how to drag objects from one container to another container while giving each container the ability to either accept or reject the drop based on the type of object being dropped.

Functions, classes, methods, and events

The sample program that I will explain in this lesson will use the following functions, classes, methods, and events plus others as well:

- flash.utils.getQualifiedClassName top level function
- MouseEvent class
    - MouseDown event
- DragEvent class
    - dragDrop event
    - dragEnter event
- DragManager class
    - acceptDragDrop method

    - doDrag method

- DragSource class

    - addData method
    - hasFormat method

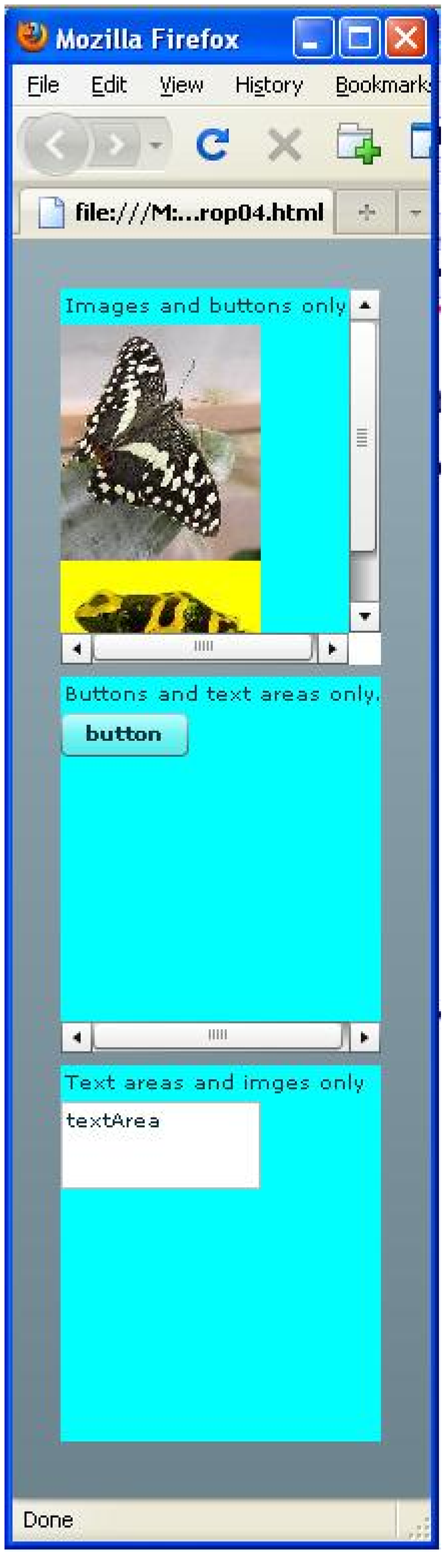
Preview

In this lesson, I will explain how to drag and drop components from one container into another container while giving the receiving container the ability to accept or reject the drop on the basis of the type of object being dragged.

Program output at startup

Figure 1 shows the program output at startup.
Program output at startup.

Figure


Mozilla Firefox
File Edit View History Bookmarks
file:///M:...rop04.html
Images and buttons only
Buttons and text areas only.
button
Text areas and imges only
textArea
Done

Program output at startup.

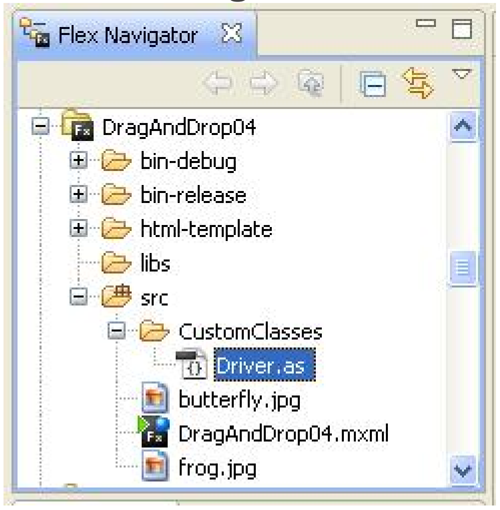
Three Canvas objects in a VBox object

The program places three **Canvas** objects in a **VBox** object as shown in Figure 1. It places two Image objects in one of the **Canvas** objects. The Image objects are populated with the contents of the following two image files, which are shown in the project file structure in Figure 2.

- butterfly.jpg
- frog.jpg

Project file structure.

Figure



Project file structure.

The image of the project file structure shown in Figure 2 was taken from the Flex Builder 3 Navigator pane.

The program also places a **Button** object in the middle **Canvas** object shown in Figure 1, and places a **TextArea** object in the bottom **Canvas** object.

Draggable objects

The two Image objects as well as the **Button** object and the **TextArea** object are all draggable.

The Canvas objects are labeled

The program places a **Label** object at the top of each **Canvas** object. The labels are for information purposes and are not draggable.

Allowable drop zones

Each of the four draggable objects can be dragged and dropped within two of the three **Canvas** objects so long as the mouse pointer is inside the **Canvas** object when the drop occurs.

None of the draggable objects can be dropped in all three of the **Canvas** objects. The label at the top of each **Canvas** object tells which types of objects can be dropped into that particular **Canvas** object.

Protection on the left and top

An object may not be dropped in such a way that it protrudes outside the left edge or the top of a **Canvas** object. If an attempt is made to do so when an object is being dragged to a new location within the same **Canvas** object, it simply returns to its original position.

If this happens when the object is being dragged into a different **Canvas** object, it assumes the same relative position in the new Canvas object that it previously occupied in the **Canvas** object from which it was dragged.

Scrollbars magically appear

If the object is dropped such that it protrudes outside the right side or the bottom of the **Canvas** object, scroll bars automatically appear on the **Canvas** object.

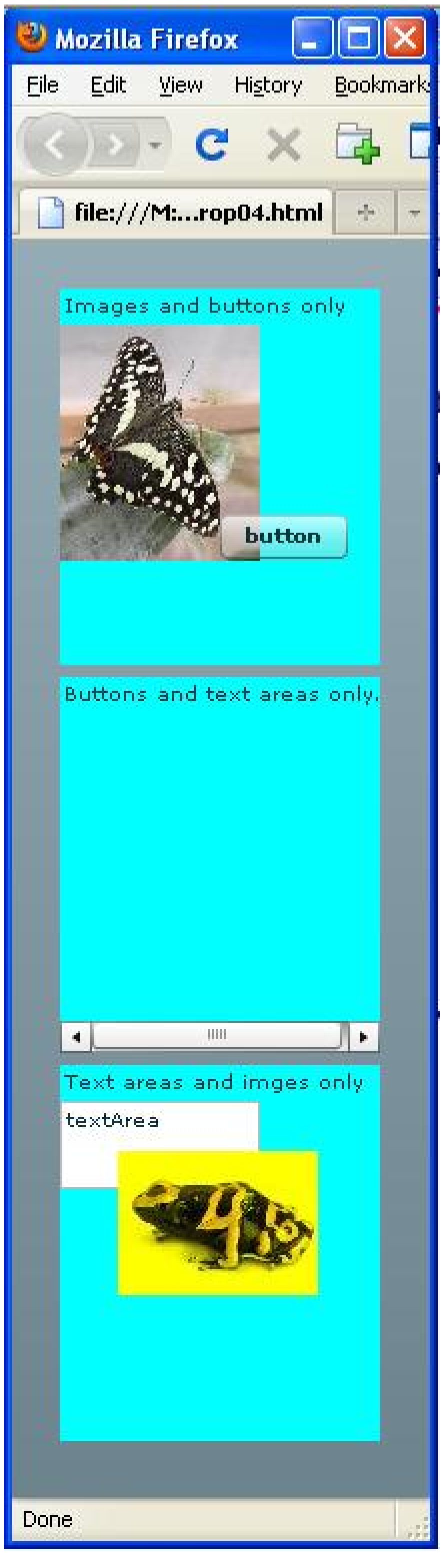
Allowable object types in the different Canvas objects

The following list shows the types of draggable objects that can be dropped into each of the Canvas objects:

- Top canvas: images and buttons only.
- Middle canvas: buttons and text areas only.
- Bottom canvas: text areas and images only.

Results of dragging objects

Figure 3 shows the results of dragging the button into the top **Canvas** object and dragging an image into the bottom **Canvas** object. You cannot drop the **TextArea** object into the top canvas, an image into the middle canvas, or the button into the bottom canvas.
Results of dragging objects.
Figure

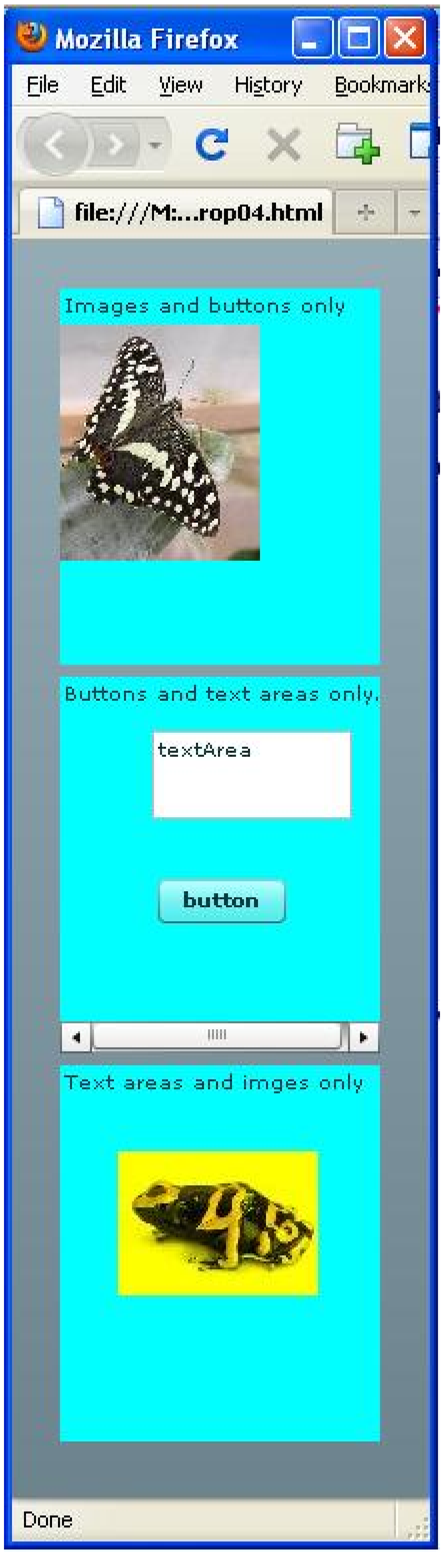

Mozilla Firefox
File Edit View History Bookmarks
file:///M:...rop04.html
Images and buttons only
button
Buttons and text areas only.
Text areas and imges only
textArea
Done

Results of dragging objects.

Figure 4 shows the result of dropping the **TextArea** object into the middle **Canvas** object.

Result of droping the TextArea into the middle Canvas.

Figure


Mozilla Firefox
File Edit View History Bookmarks
file:///M:...rop04.html
Images and buttons only
Buttons and text areas only.
textArea
button
Text areas and imges only
Done

Result of droping the
TextArea into the

middle Canvas.

Discussion and sample code

Will explain in fragments

I will explain the code for this program in fragments. Complete listings of the MXML code and the ActionScript code are provided in Listing 13 and Listing 14 near the end of the lesson.

The MXML code

The MXML code is shown in Listing 1 and also in Listing 13 for your convenience.

Example:
The MXML code.

```
<?xml version="1.0" encoding="utf-8"?>
<!--DragAndDrop04
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">
  <cc:Driver/>

</mx:Application>
```

As is often the case in this series of tutorial lessons, the MXML file is very simple because the program was coded almost entirely in ActionScript. The MXML code simply instantiates an object of the **Driver** class. From that point forward, the behavior of the program is controlled by ActionScript code.

The ActionScript code

The beginning of the Driver class

The **Driver** class begins in Listing 2.

Example:
Beginning of the Driver class.

```
package CustomClasses{
  import flash.events.MouseEvent;
  import flash.utils.getQualifiedClassName;

  import mx.containers.Canvas;
  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Image;
  import mx.controls.Label;
  import mx.controls.TextArea;
  import mx.core.DragSource;
  import mx.core.UIComponent;
  import mx.events.DragEvent;
  import mx.events.FlexEvent;
  import mx.managers.DragManager;


//==================================================
====//
```

```
  public class Driver extends VBox {
    private var button:Button = new Button();
    private var butterfly:Image = new Image();
    private var frog:Image = new Image();
    private var textArea:TextArea = new
TextArea();
    private var canvasA:Canvas = new Canvas();
    private var canvasB:Canvas = new Canvas();
    private var canvasC:Canvas = new Canvas();
    private var labelA:Label = new Label();
    private var labelB:Label = new Label();
    private var labelC:Label = new Label();
    private var localX:Number;
    private var localY:Number;

    public function Driver(){//constructor

      //Put a label at the top of each Canvas
object.
      labelA.text = "Images and buttons only";
      labelB.text = "Buttons and text areas
only.";
      labelC.text = "Text areas and imges only";
      canvasA.addChild(labelA);
      canvasB.addChild(labelB);
      canvasC.addChild(labelC);

      //Add the Canvas objects to the VBox object
      addChild(canvasA);
      addChild(canvasB);
      addChild(canvasC);

      //Embed the image files in the SWF file.
      [Embed("butterfly.jpg")]
      var butterflyA:Class;
      [Embed("frog.jpg")]
      var frogA:Class;

      //Load the images from the embedded image
files
      // into the Image objects.
      butterfly.load(butterflyA);
      frog.load(frogA);

      //Put some text on the button and in the
TextArea.
      button.label = "button";
      textArea.text = "textArea";

      //Add the components to the Canvas objects.
      canvasA.addChild(butterfly);
      canvasA.addChild(frog);
      canvasB.addChild(button);
      canvasC.addChild(textArea);

      //Register an event handler that will be
executed
      // whcn the canvas and its children are
fully
      // constructed.

this.addEventListener(FlexEvent.CREATION_COMPLETE,

completeHandler);
    } //end constructor
```

Nothing new here

There is nothing new in Listing 2 that you haven't learned about in previous lessons.

A creationComplete event handler

The last statement in Listing 2 registers a **CREATION_COMPLETE** event handler on the **VBox** object. The code in the handler is executed when the **VBox** object and all of its children have been fully created. The event handler begins in Listing 3.

Example:
Beginning of the creationComplete event handler.

```
    private function completeHandler(

event:mx.events.FlexEvent):void{
      //Make the Canvas objects visible.

canvasA.setStyle("backgroundColor",0x00FFFF);

canvasB.setStyle("backgroundColor",0x00FFFF);

canvasC.setStyle("backgroundColor",0x00FFFF);

      //Set the width and height of the canvas
objects
      // based on the dimensions of butterfly.
      canvasA.width = 1.6*butterfly.width;
      canvasA.height = 1.6*butterfly.height;
      canvasB.width = 1.6*butterfly.width;
      canvasB.height = 1.6*butterfly.height;
      canvasC.width = 1.6*butterfly.width;
      canvasC.height = 1.6*butterfly.height;

      //Reduce the width of the textArea to less
than
      // its default width.
      textArea.width = butterfly.width;
      //Move the images, the button, and the text 
area
      // below the label
      butterfly.y = labelA.height;
      frog.y = labelA.height + butterfly.height;
      button.y = labelB.height;
      textArea.y = labelC.height;
```

Straightforward code

The code in Listing 3 is straightforward and you shouldn't have any difficulty understanding it based on what you have learned in previous lessons.

Register a mouseDown event listener on each draggable object

Listing 4 registers the same **MOUSE_DOWN** event listener on each draggable object. This is the beginning of the process that causes the two **Image** objects, the **Button** object, and the **TextArea** object to be draggable.

Example:
Register a mouseDown event listener on each draggable object.

```
butterfly.addEventListener(MouseEvent.MOUSE_DOWN, 

mouseDownHandler);
      frog.addEventListener(MouseEvent.MOUSE_DOWN, 

mouseDownHandler);

button.addEventListener(MouseEvent.MOUSE_DOWN,
mouseDownHandler);

textArea.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);
```

There is nothing new or unique about the registration code in Listing 4. You have seen code like that in almost every lesson. The material that is unique to the drag and drop process is the code in the **mouseDown** event handler method that I will discuss next.

The beginning of the mouseDown event handler

I will put the explanation of the **creationComplete** event handler on hold while I explain the **mouseDown** event handler. I will return to the **creationComplete** event handler later.

The **mouseDown** event handler that is registered on the four draggable objects begins in Listing 5. This method is executed whenever any one of those objects dispatches a **mouseDown** event, and that is the beginning of the drag and drop process.

Example:
The beginning of the mouseDown event handler.

```
    private function mouseDownHandler(

event:MouseEvent):void{
      this.localX = event.localX;
      this.localY = event.localY;
```

Listing 5 begins by getting and saving the coordinates of the mouse pointer, relative to the upper-left corner of the object that dispatches the **mouseDown** event. As you saw in the earlier lesson titled **Drag and Drop Basics** , this information will be used later to properly position the dropped image in the **Canvas** object.

Get the name of the class from which the drag initiator was instantiated

The code in Listing 6 is completely new to this lesson.

Example:
Get the name of the class from which the drag initiator was instantiated.

```
      var className:String =
getQualifiedClassName(

event.target);
```

The Adobe documentation refers to the object that dispatches the **mouseDown** event to start the drag and drop process as the **drag initiator** .

Call the standalone getQualifiedClassName function

Listing 6 calls the function named **getQualifiedClassName** to get a string containing the name of the class from which the drag initiator object was instantiated. Note that this is a stand-alone function in the **flash.utils** package.

Non-unique strings

Different classes return different strings but the strings are not unique. In other words, two or more classes may return the same string. For example, several different Flex components return the same string as the string returned by the **Button** component.

The three draggable component types used in this program return the following **strings** :

- **Image** returns "mx.controls::Image"
- **Button** returns "mx.controls::Button"
- **TextArea** returns "mx.core::UITextField"

Save the string

The string returned by the object that dispatched the **mouseDown** event is saved in the variable named **className** in Listing 6. This string value will be used later to identify the type of component that dispatched the **mouseDown** event.

Some of the steps in the process...

When the object being dragged enters one of the **Canvas** objects, the **Canvas** object dispatches a **dragEnter** event, which is a subtype of the class **DragEvent** . The **dragEnter** event handler receives an object of type **DragEvent** , which encapsulates an object of the class **DragSource** .

The DragSource object

The **DragSource** object encapsulates a reference to the drag initiator object *(the object that dispatched the **mouseDown** event)* . It also encapsulates a format string that can be used to identify the drag initiator object. In this program, that string is used by a particular **Canvas** object to determine if it will accept a drop by the drag initiator object.

Populate a new DragSource object with the drag initiator and a format string

Listing 7 instantiates a new **DragSource** object and populates it with the drag initiator and a format string that identifies the type of the drag initiator. The format string is based on the class from which the drag initiator was instantiated using the information in the **className** variable from Listing 6 in the conditional clause.

Example:
Populate a new DragSource object with the drag initiator and a format string.

```
      var dragSource:DragSource = new
DragSource();

      if(className == "mx.controls::Image"){
        dragSource.addData(UIComponent(

event.currentTarget),"imageObj");
      }else if(className == "mx.controls::Button")
{
        dragSource.addData(UIComponent(

event.currentTarget),"buttonObj");
      }else if(className ==
"mx.core::UITextField"){
        dragSource.addData(UIComponent(

event.currentTarget),"textAreaObj");
      } //end else if
```

Can the drag initiator object be dropped?

A **Canvas** object that examines the format string from the **DragSource** object later in the program will use the format string to determine if it is willing to allow the drag initiator object to be dropped on it.

Cast the current target to type UIComponent

The term **event.currentTarget** is a reference to the object that dispatched the **mouseDown** event, which is the drag initiator. However, when the **currentTarget** is extracted from the **MouseEvent** object, it is returned as type **Object** . In order for it to be suitable for use as the drag initiator, it must be cast to type **UIComponent** .

Relationships among the strings

I could have simply used the strings from the above list as the format strings, but I decided to create new strings to show that they are really independent of one another.

The relationships that I created between the strings in the above list and the format strings is shown below:

- **Image** returns "mx.controls::Image" -- "imageObj"
- **Button** returns "mx.controls::Button" -- "buttonObj"
- **TextArea** returns "mx.core::UITextField" -- "textAreaObj"

Initiate the drag and drop operation

Listing 8 calls the static method named **doDrag** of the **DragManager** class to initiate the drag and drop operation.

Example:
Initiate the drag and drop operation.

```
DragManager.doDrag(UIComponent(event.currentTarget
),
dragSource,event);
    }//end mouseDownHandler
```

I explained the use of this method in the previous lesson titled **Drag and Drop Basics** so I will refer you back to that lesson for a detailed explanation.

Register a different dragEnter event handler on each Canvas object

Returning to the discussion of the **creationComplete** event handler, Listing 9 registers a different **dragEnter** event handler on each **Canvas** object.

Example:
Register a different dragEnter event handler on each Canvas object.

```
      //Register a different dragEnter event
handler on
      // each Canvas object to make it possible
for each
      // Canvas object to accept only two of the
three
      // types of components for dropping.

canvasA.addEventListener(DragEvent.DRAG_ENTER,

enterHandlerA);

canvasB.addEventListener(DragEvent.DRAG_ENTER,

enterHandlerB);

canvasC.addEventListener(DragEvent.DRAG_ENTER,
enterHandlerC);
```

Dispatching dragEnter events

As you learned in the earlier lesson titled **Drag and Drop Basics** , when the user moves the drag proxy over another component, that component dispatches a **dragEnter** event.

If a **dragEnter** event handler has been registered on that component, the handler method is executed. If the code in the event handler **accepts** the drag, it becomes the drop target and receives **dragOver** , **dragExit** , and **dragDrop** events.

I will explain the code in the **dragEnter** event handlers later. Right now, let's look at the remainder of the **creationComplete** event handler.

Register the same dragDrop event handler on all three Canvas objects

Listing 10 registers the same **dragDrop** event handler on each of the three **Canvas** objects

Example:
Register the same dragDrop event handler on all three Canvas objects.

```
      //Register the same dragDrop event handler
on all
      // three Canvas objects.

canvasA.addEventListener(DragEvent.DRAG_DROP,

dropHandler);

canvasB.addEventListener(DragEvent.DRAG_DROP,
dropHandler);

canvasC.addEventListener(DragEvent.DRAG_DROP,

dropHandler);

    } //end completeHandler
```

The code in Listing 10 will cause another event handler to be called on a **Canvas** object after it dispatches a **dragEnter** event and accepts the drag.

Listing 10 also signals the end of the **creationComplete** event handler.

The dragEnter event handlers

A different **dragEnter** event handler was registered on each **Canvas** object in Listing 9.

Each of the **dragEnter** event handlers is executed when the dragged component enters the **Canvas** object on which the handler is registered. The event handlers decide whether or not to accept a drop on the basis of the format string associated with the type of object being dragged. Note that each **Canvas** object will accept two of the three types of objects.

Will only explain one of the dragEnter event handlers

Because of the similarity of the three event handlers, I will explain only one of them. You can view the code for all three in Listing 14 near the end of the lesson. Listing 11 shows a **dragEnter** event handler for the top **Canvas** object shown in Figure 1.

Example:
A dragEnter event handler for the top Canvas object.

```
    //This dragEnter event handler causes the
canvas to
    // accept images and buttons.
    private function
enterHandlerA(event:DragEvent):void{
      if ((event.dragSource.hasFormat("imageObj"))
||

(event.dragSource.hasFormat("buttonObj"))){

          DragManager.acceptDragDrop(

Canvas(event.currentTarget));
      } //end if
    } //end enterHandler
```

Confirm the correct format string

The event handler shown in Listing 11 causes the **Canvas** object to accept only images and buttons. The only difference between the code in Listing 11 and similar code in the earlier lesson titled **Drag and Drop Basics** is the use of the logical-or (||) operator in Listing 11 to accept either of the two string instead of just one.

The code in Listing 11 checks to confirm that the format string in the **DragSource** object matches either "imageObj" or "buttonObj" *(see Listing 7)* . If so, it calls the static **acceptDragDrop** method on the **DragManager** class, passing a reference to itself as a parameter in the method call.

Accept the dragged object

The call to the **acceptDragDrop** method notifies the **DragManager** that the **Canvas** object is willing to accept the contents of the **DragSource** object being dropped onto itself.

The dragDrop event handler

The **dragDrop** event handler was registered on all three **Canvas** objects in Listing 10. This method is executed after a **Canvas** object accepts the drag and the user releases the mouse button while the drag proxy is over the **Canvas** .

The **dragDrop** event handler method is shown in its entirety in Listing 12.

Example:
The dragDrop event handler.

```
    //Execute the dragDrop event handler to drop
the
    // object in its new location. Compensate for
the
    // fact that the mouse pointer is not at the
    // upper-left corner of the object when the
drag is
    // initiated. Don't allow the image to be
dragged off
    // the left side of the canvas or off the top
of the
    // canvas. See more information about this in
the
    // comments at the top.
    private function
dropHandler(event:DragEvent):void{

      //Add the drag initiator to the new
container.
      // Note that it is not necessary to first
remove it
      // from its old container.
      //Also note that the z-axis index is lost in
this
```

```
      // operation. When an object is dropped on
top of
      // another object in the new container, it
stays on
      // top regardless of the original z-order of
the
      // two objects.
      //The original z-order has no meaning when
you drag
      // objects into a canvas from several othr
Canvas
      //

event.currentTarget.addChild(event.dragInitiator);

      //Position the dragInitiator in the Canvas
object
      // based on the mouse coordinates at the
drop and
      // the mouse coordinates relative to the
upper-
      // left corner of the drag initiator at the
      // start of the drag.
      //Compute the correct position for the
upper-left
      // corner of the dropped object.
      //If you attempt to drop an object so that
it
      // protrudes out of the left side or the top
of
      // the canvas, the drag and drop operation
is
      // simply aborted.
      var cornerX:Number =
(Canvas(event.currentTarget).
                                         mouseX) -
localX;
```

```
      var cornerY:Number =
(Canvas(event.currentTarget).
                                         mouseY) -
localY;
      if((cornerX > 0.0) && (cornerY > 0.0)){
        event.dragInitiator.x = cornerX;
        event.dragInitiator.y = cornerY
      } //end if
    } //end dropHandler
    //--------------------------------------------------
------//

  } //end class
} //end package
```

The new code

The only real difference between the code in Listing 12 and the similar event handler in the earlier lesson titled **Drag and Drop Basics** is the first statement in Listing 12 that adds the dragged object as a child of the **Canvas** object.

What you learned in the earlier lesson in conjunction with the comments in Listing 12 should suffice and no further explanation of this method should be necessary.

The end of the program

Listing 12 also signals the end of the **Driver** class, the end of the package, and the end of the program.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 13 and Listing 14. Use that code to create a Flex project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the MXML code and the ActionScript code are provided below.

Example:
The MXML code for the program named DragAndDrop04.

```
<?xml version="1.0" encoding="utf-8"?>
<!--DragAndDrop04
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">
  <cc:Driver/>

</mx:Application>
```

Example:
The ActionScript code for the program named DragAndDrop04.

```
/*DragAndDrop04
```

```
/ DragAndDrop04

Illustrates how to drag components from one
container
to another while giving the receiving container
the
ability to accept or reject the drop on the basis
of the
type of object being dragged.

Places three Canvas objects in a VBox object.

Places two images in one of the Canvas objects:
butterfly.jpg
frog.jpg

Places a Button object in a second Canvas object.

Places a TextArea object in the third Canvas
object.

All four of above objects are draggable.

Places a label at the top of each Canvas object,
but the
labels are not draggable.

Any of the four draggable objects can be dragged
and
dropped anywhere within two of the three Canvas
objects
so long as the mouse pointer is inside the Canvas
object.

None of the objects can be dropped in all three of
the
Canvas objects.
```

```
If an object is dropped so that it protrudes
outside the
left edge or the top of the Canvas object when
being
dragged to a new location within the same Canvas
object,
it simply returns to its original position. If
this
happens when the object is being dragged into a
different
Canvas object, it assumes the same relative
position in
the new Canvas object that it previously occupied
in the
Canvas object from which it was dragged.

If the dragged object is dropped such that it
protrudes
outside the right side or the bottom of the Canvas
object, scroll bars automatically appear on the
Canvas
object.

The size of the canvas is based on the size of the
butterfly image so that other images can be
substituted
for my images when the program is recompiled so
long as
the file names and paths are correct.
**************************************************
*******/
package CustomClasses{
  import flash.events.MouseEvent;
  import flash.utils.getQualifiedClassName;


  import mx.containers.Canvas;
  import mx.containers.VBox;
  import mx.controls.Button;
  import mx.controls.Image;
  import mx.controls.Label;
  import mx.controls.TextArea;
  import mx.core.DragSource;
  import mx.core.UIComponent;
  import mx.events.DragEvent;
  import mx.events.FlexEvent;
  import mx.managers.DragManager;


//================================================
====//

  public class Driver extends VBox {
    private var button:Button = new Button();
    private var butterfly:Image = new Image();
    private var frog:Image = new Image();
    private var textArea:TextArea = new
TextArea();
    private var canvasA:Canvas = new Canvas();
    private var canvasB:Canvas = new Canvas();
    private var canvasC:Canvas = new Canvas();
    private var labelA:Label = new Label();
    private var labelB:Label = new Label();
    private var labelC:Label = new Label();
    private var localX:Number;
    private var localY:Number;

    public function Driver(){//constructor

      //Put a label at the top of each Canvas
object.
      labelA.text = "Images and buttons only";

      labelB.text = "Buttons and text areas
only.";
      labelC.text = "Text areas and imges only";
      canvasA.addChild(labelA);
      canvasB.addChild(labelB);
      canvasC.addChild(labelC);

      //Add the Canvas objects to the VBox object
      addChild(canvasA);
      addChild(canvasB);
      addChild(canvasC);

      //Embed the image files in the SWF file.
      [Embed("butterfly.jpg")]
      var butterflyA:Class;

      [Embed("frog.jpg")]
      var frogA:Class;

      //Load the images from the embedded image
files
      // into the Image objects.
      butterfly.load(butterflyA);
      frog.load(frogA);

      //Put some text on the button and in the
text area.
      button.label = "button";
      textArea.text = "textArea";

      //Add the components to the Canvas objects.
      canvasA.addChild(butterfly);
      canvasA.addChild(frog);
      canvasB.addChild(button);
      canvasC.addChild(textArea);

      //Register an event handler that will be

executed
      // whcn the canvas and its children are
fully
```

```
      // constructed.

this.addEventListener(FlexEvent.CREATION_COMPLETE,

completeHandler);
    } //end constructor
    //--------------------------------------------------
------//

    //This handler method is executed when the
Canvas and
    // its children have been fully created.
    private function completeHandler(

event:mx.events.FlexEvent):void{
      //Make the Canvas objects visible.

canvasA.setStyle("backgroundColor",0x00FFFF);

canvasB.setStyle("backgroundColor",0x00FFFF);

canvasC.setStyle("backgroundColor",0x00FFFF);

      //Set the width and height of the canvas
objects
      // based on the dimensions of butterfly.
      canvasA.width = 1.6*butterfly.width;
      canvasA.height = 1.6*butterfly.height;
      canvasB.width = 1.6*butterfly.width;
      canvasB.height = 1.6*butterfly.height;
      canvasC.width = 1.6*butterfly.width;
      canvasC.height = 1.6*butterfly.height;

      //Reduce the width of the textArea to less

than
      // its default width.
      textArea.width = butterfly.width;
      //Move the images, the button, and the text
area
      // below the label
      butterfly.y = labelA.height;
      frog.y = labelA.height + butterfly.height;
      button.y = labelB.height;
      textArea.y = labelC.height;

      //Register event listeners to support drag
and drop
      // operations on both images, the button,
and the
      // text area with the canvas as the drag
target.

butterfly.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);
      frog.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);

button.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);

textArea.addEventListener(MouseEvent.MOUSE_DOWN,

mouseDownHandler);

      //Register a different dragEnter event
handler on
      // each Canvas object to make it possible

for each
      // Canvas object to accept only two of the
three
```

```
      // types of components for dropping.

canvasA.addEventListener(DragEvent.DRAG_ENTER,

enterHandlerA);

canvasB.addEventListener(DragEvent.DRAG_ENTER,

enterHandlerB);

canvasC.addEventListener(DragEvent.DRAG_ENTER,

enterHandlerC);

      //Register the same dragDrop event handler
on all
      // three Canvas objects.

canvasA.addEventListener(DragEvent.DRAG_DROP,

dropHandler);

canvasB.addEventListener(DragEvent.DRAG_DROP,

dropHandler);

canvasC.addEventListener(DragEvent.DRAG_DROP,

dropHandler);

    } //end completeHandler
    //--------------------------------------------
------//


    // This event handler initiates the drag-and-
drop \
    // operation for the image that dispatches the
    // mouseDown event.
    private function mouseDownHandler(

event:MouseEvent):void{

      //Save the location of the mouse within the
object
      // being dragged. This information will be
used
      // later to properly position the dropped
image in
      // the drop target.
      this.localX = event.localX;
      this.localY = event.localY;

      //The drag initiator is the object that
dispatched
      // this mouseDown event. Get a string
containing
      // the name of the class from which that
object was
      // instantiated. For the components used in
this
      // program, the possible strings are:
      // Image - "mx.controls::Image"
      // Button - "mx.controls::Button"
      // TextAra - "mx.core::UITextField"
      //Note, the following function is in the
      // flash.utils package.
      var className:String =
getQualifiedClassName(

event.target);


      //Populate a new DragSource object with the
drag
      // initiator and a format string based on
```

```
the class
      // from which the drag initiator was
instantiated.
      //The format string will be used later to
decide
      // if a particular Canvas object is willing
to
      // allow a particular type of object to be
dropped
      // on it.
      //Note that when the target of the mouseDown
event
      // is used as the drag initiator, it must be
cast
      // to type UIComponent.
      var dragSource:DragSource = new
DragSource();
      if(className == "mx.controls::Image"){
        dragSource.addData(UIComponent(

event.currentTarget),"imageObj");
      }else if(className == "mx.controls::Button")
{
        dragSource.addData(UIComponent(

event.currentTarget),"buttonObj");
      }else if(className ==
"mx.core::UITextField"){
        dragSource.addData(UIComponent(

event.currentTarget),"textAreaObj");
      } //end else if

      //Initiate the drag and drop operation.


DragManager.doDrag(UIComponent(event.currentTarget
),
dragSource,event);
    }//end mouseDownHandler
    //-------------------------------------------------
------//

    //Each of the following dragEnter event
handlers is
    // executed when the dragged omponent enters
the
    // Canvas object on which the handlr is
registered.
    // The event handlers decide whether or not to
accept
    // a drop on the basis ofthe format string
associated
    // with the type of object being dragged. Note
that
    // each Canvas ovject will accept two of the
three
    // types of objects.

    //This dragEnter event handler causes the
canvas to
    // accept images and buttons.
    private function
enterHandlerA(event:DragEvent):void{
      if ((event.dragSource.hasFormat("imageObj"))
||

(event.dragSource.hasFormat("buttonObj"))){

          DragManager.acceptDragDrop(


Canvas(event.currentTarget));
      } //end if
    } //end enterHandler
    //---------------------------------------------
------//

    //This dragEnter event handler causes the
canvas to
    // accept textAreas and buttons.
    private function
enterHandlerB(event:DragEvent):void{
      if
((event.dragSource.hasFormat("textAreaObj")) ||

(event.dragSource.hasFormat("buttonObj"))){

          DragManager.acceptDragDrop(

Canvas(event.currentTarget));
      } //end if
    } //end enterHandler
    //---------------------------------------------
------//

    //This dragEnter event handler causes the
canvas to
    // textAreas and images.
    private function
enterHandlerC(event:DragEvent):void{
      if
((event.dragSource.hasFormat("textAreaObj")) ||

(event.dragSource.hasFormat("imageObj"))){

          DragManager.acceptDragDrop(

Canvas(event.currentTarget));

      } //end if
    } //end enterHandler
    //---------------------------------------------
```

```
------//

    //Execute the dragDrop event handler to drop
the
    // object in its new location. Compensate for
the
    // fact that the mouse pointer is not at the
    // upper-left corner of the object when the
drag is
    // initiated. Don't allow the image to be
dragged off
    // the left side of the canvas or off the top
of the
    // canvas. See more information about this in
the
    // comments at the top.
    private function
dropHandler(event:DragEvent):void{

      //Add the drag initiator to the new
container.
      // Note that it is not necessary to first
remove it
      // from its old container.
      //Also note that the z-axis index is lost in
this
      // operation. When an object is dropped on
top of
      // another object in the new container, it
stays on
      // top regardless of the original z-order of
the
      // two objects.
      //The original z-order has no meaning when

you drag
      // objects into a canvas from several othr
Canvas
```

```
      //

event.currentTarget.addChild(event.dragInitiator);

      //Position the dragInitiator in the Canvas
object
      // based on the mouse coordinates at the
drop and
      // the mouse coordinates relative to the
upper-
      // left corner of the drag initiator at the
      // start of the drag.
      //Compute the correct position for the
upper-left
      // corner of the dropped object.
      //If you attempt to drop an object so that
it
      // protrudes out of the left side or the top
of
      // the canvas, the drag and drop operation
is
      // simply aborted.
      var cornerX:Number =
(Canvas(event.currentTarget).
                                         mouseX) -
localX;
      var cornerY:Number =
(Canvas(event.currentTarget).
                                         mouseY) -
localY;
      if((cornerX > 0.0) && (cornerY > 0.0)){
        event.dragInitiator.x = cornerX;
        event.dragInitiator.y = cornerY
      } //end if

    } //end dropHandler
    //---------------------------------------------
------//
```

```
  } //end class
} //end package
```

Miscellaneous

Note: Housekeeping material

- Module name: Dragging Objects between Containers
- Files:
    - ActionScript0142\ActionScript0142.htm
    - ActionScript0142\Connexions\ActionScriptXhtml0142.htm

-end-

Animation Fundamentals
Learn something about how the Flash Player behaves, and learn to use that behavior to write animation projects. Also learn about two different time bases that you can use to control the timing of your ActionScript animation projects.

Note: Click Animation01 , Animation07 , or Animation01A to run the ActionScript programs from this lesson. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

Note: All references to ActionScript in this lesson are references to version 3 or later.

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

Several ways to create and launch ActionScript projects

There are several ways to create and launch projects written in the ActionScript programming language. Many, but not all of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript projects.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. The lesson titled **Using Flex 3 in a Flex 4 World** was added later to accommodate the release of Flash Builder 4. *(See Baldwin's Flex programming website .)* You should study those lessons before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

I am going to take a very broad view of *computer animation* in this and the next few lessons. A program that causes visual images to *change over time* produces what I am referring to an animation.

Such visual changes can take on many different forms. Perhaps the most common form of animation is the movement *(change in position)* of an object over time. However, animation can involve a change in any visual attribute of an object over time, such as changes in position, color, size, opacity, orientation, etc.

A layman's view of the Flash Player

Major differences

There are some major differences between writing animation code in ActionScript and writing animation code in Java, C++, and C#.

Those differences generally have to do with the strong tie between the execution of ActionScript code and the Flash Player. *(In some cases, references to the Flash Player in this lesson may also apply to Adobe Air.)*

To understand how to write animation code in ActionScript, you must first understand a little about the Flash Player.

Why is it called a *player?*

There is a good reason that the Flash Player is referred to as a *player* . In many ways, it resembles a DVD player or a VCR player. What I mean by that is that the Flash Player extracts a series of visual images from memory and displays those images sequentially with a *(hopefully)* fixed time interval between the display of one image and the display of the next image in the sequence.

The scene is displayed once during each frame

We can't know exactly what happens at the lowest levels of the software and the display hardware. However, from the viewpoint of the ActionScript programmer, the scene described by the images stored in memory is displayed over and over even if it isn't changing.

Note: According to the *ActionScript 3.0 Bible, 2nd Edition*, by Braunstein, ActionScript 3.0 uses a concept known as *"dirty rectangles"* to avoid physically rendering rectangular groups of pixels that haven't changed.

Frames

Each new display of the scene is referred to as a *frame* . As you will see later, it is possible for the ActionScript programmer to simply accept the default rate at which frames are displayed, or to execute code to set the *frame rate* to something other than the default rate.

Not the case in Java, C++, and C#

The inherently strong tie between the images stored in memory and the sequential and repetitive display of those images by the Flash Player does not exist in Java, C++, or C# *(although it can be **simulated**)* .

Note: There is an open-source programming environment named Processing that uses Java to create an architecture very similar to the Flash Player. The Microsoft **XNA Game Studio** *(C#)* and Game Creators **Dark GDK** *(C++)* also create a similar architecture. Note, however, that these are optional add-ons to the language, which is not the case regarding ActionScript and the Flash Player.

The programmer is responsible for the display

Generally speaking, when writing code in those other languages, it is up to the programmer to write the code that determines how, if, and when the images are displayed on the screen.

While it is not unusual for the programmer to write code to cause images to be displayed in a sequential and repetitive manner in those other languages *(see the above note)* , the decision to do that is strictly up to the programmer.

Good news and bad news

Note: It is possible to *(almost)* prevent the Flash Player from displaying repetitive frames by writing code to set the frame rate to 0.01 frames per second. This is not without its problems however.

There are pros and cons to both approaches. The good news is that the behavior of the Flash Player relieves the programmer of the responsibility to display images on a sequential and repetitive basis.

The bad news is that the Flash Player deprives the programmer of the opportunity to make decisions in that regard.

Startup considerations

What happens when a swf file is loaded?

It is hard to know exactly what happens when a swf file is loaded into the Flash player. I have searched the web extensively and have been unable to find definitive information in that regard.

However, I think I understand what happens, and I will share what I believe to be true with you.

Different ways to write ActionScript 3 programs

You can write ActionScript 3 programs by creating AS3 or ActionScript projects using **Adobe's Flex Builder 3** , **Adobe's Flash Builder 4** , or using the free open source FlashDevelop software.

Note: There are other ways to write ActionScript 3 projects as well.

A skeleton ActionScript source code file

With all three tools, the creation of an ActionScript project causes a skeleton ActionScript source code file with an extension of *.as* to be created. This file must contain a public class definition with a class name that matches the name of the file exclusive of the extension.

The name of the ActionScript source code file

The default name of the source code file differs for the three development tools, but you can rename it if you wish for all three tools. However, you must also be sure to rename the class definition *(and its constructor)* so that the name of the class matches the name of the source code file.

A *release build*

In all three cases, when you create a *release build* for the project, you end up with a folder containing an swf file, an html file, and some other folders and files as a minimum.

The name of the swf file

The name of the swf file also varies among the different development tools and it is best not to change it. It is referenced inside the html file and possibly inside some of the other files as well, so changing the name of the swf file would require you to make corresponding changes in one or more other files.

The name of the html file

The name of the html file also varies among the different development tools but it appears that you can change it with no ill effects.

What happens when the html file is opened in a browser?

Here is what I believe happens when the html file is opened in a browser that has the Flash Player plug-in installed.

Note: This may be entirely wrong from a detailed technical viewpoint, but I will present it for your consideration because it seems to describe what actually happens.

The swf file is opened in the Flash Player

The code inside the html file causes the swf file to be opened in the Flash Player plug-in that has been installed in that browser.

The swf file contains the compiled class definition for the class defined in the source code file mentioned earlier plus a number of other things, including the name of that class. *(You can probably think of a swf file as being similar to a zip file or a Java JAR file; a file that encapsulates other files.)*

An object of the class is instantiated

The Flash Player plug-in extracts the name of the class and calls the constructor for that class to instantiate an object of that class.

The constructor is executed

All of the code contained in the constructor plus all of the code called by code in the constructor, plus all of the code called by that code, etc., is executed.

Objects of type *DisplayObject* are displayed

If any of that code instantiates objects that derive from the **DisplayObject** class and adds those objects to the *display list* , those objects will be displayed during the next frame.

The current contents of the display list are displayed during each frame

If all of that code terminates without doing something to cause the contents of the display list to be modified or to cause the attributes of those display objects to be modified in the future, they will continue to be displayed once during each frame. In that case, the display will appear to be static insofar as the user is concerned.

Note: Note that some objects, such as **Button** objects, inherently know how to modify their own attributes under certain circumstances, such as being rolled over or clicked by the mouse.

Additional code may be executed in the future

The code that is executed and caused to be executed by the constructor may do something to ensure the future execution of additional code, *(such as registering TIMER or ENTER_FRAME event listeners).*

Code that is executed in the future may modify the display list, may modify the attributes of existing objects on the display list, may instantiate new display objects and add them to the display list, etc. Such changes will be reflected in the visual screen display when they occur.

Frame-to-frame changes in the display

As a result of code that is executed in the future, the display list may change on a frame-to-frame basis causing the physical display to also change on a frame-to-frame basis. In that case, the display won't appear to be static insofar as the user is concerned.

That will be the case for the animation projects that I explain in this lesson.

Time base considerations

In many cases when writing animation code, it is appropriate to use a stable time base to control progress through the program. There are at least two different ways to access a time base when writing ActionScript code:

- Listen for **ENTER_FRAME** events
- Listen for **TIMER** events fired by an object of the **Timer** class

Not independent approaches

Note, however, that these two approaches are not independent of one another. It is easy to write code that uses the real-time clock to show that simply changing the frame rate will cause the **TIMER** event rate to error from its specified value.

Also, unless the method named **updateAfterEvent** is called in the **TIMER** event handler, it does very little good to use a **Timer** object with a fast event rate in an attempt to produce smooth animations if the frame rate is slower than the **Timer** event rate. Without a call to that method in the event handler, changes made to the visual state of the images in accordance with the **TIMER** event rate will only appear on the screen at the slower frame rate.

Wheels turning backwards

When I was a child, I often went to the movie theatre on Saturday afternoon to watch grade-B western movies featuring stars like Hopalong Cassidy, Roy Rogers, Red Ryder, and others.

There was almost always a chase scene in which the bandits were chasing a stage coach. As a child, I could never figure out why it often looked like the wheels *(with spokes)* on the stage coach were turning backwards.

An artifact of sampling theory

Now that I understand sampling theory, I also understand what caused the wheels to turn backwards when the stage coach was moving forward.

However, an explanation of the phenomenon is beyond the scope of this lesson. *(That may make a good topic for a future lesson.)*

Note: It wouldn't be too difficult to write an ActionScript project to demonstrate this phenomenon by varying the event rate of a **Timer** object relative to the frame rate of the Flash Player, but I don't have time to do that right now.

An interaction with the frame rate

Suffice it at this point to say that the phenomenon results from an interaction between the frame rate of the movie and the speed of motion of the spokes on the wheel. *(Click here for a demonstration of the phenomenon.)*

Other dependencies

As you will see later, there are other dependencies between the **Timer** event rate and the frame rate that aren't so easy to explain.

Listen for ENTER_FRAME events

Process an event for each new display frame

According to Braunstein *(see Resources), "All display objects broadcast **Event.ENTER_FRAME** events before every frame is drawn, making them ideal timing beacons for animation."*

Note: Unfortunately, Braunstein also implies at several locations i n his excellent book that the accuracy of the repetition rate for **ENTER_FRAME** events may leave a lot to be desired .

We can define and register a listener object that will cause code to be executed each time such an event is fired. Thus, we can use the sequence of **ENTER_FRAME** events as a time base with which to control the progress of our ActionScript programs.

A simple AS3 project

Listing 16 provides the source code for the **Main** class in a very simple AS3 project *(not a Flex project)* named **TimeBase01** that illustrates the processing of **ENTER_FRAME** events for the purpose of measuring the average elapsed time between such events.

The average frame rate

This project computes and displays the average frame rate for each of five consecutive sets of 200 frames. *(The project also shows how to set the frame rate to something other than the default value.)*

Figure 1 shows the text output produced by a single run of this project in *debug* mode on a Windows Vista system.
Default frame rate for Flash Player 10.

Figure

```
1271890629505
1271890636155 200 30.075187969924812
1271890642822 400 29.99850007499625
1271890649487 600 30.007501875468865
1271890656153 800 30.003000300030006
1271890662823 1000 29.98500749625187
```

Default frame rate for Flash Player 10.

The elapsed time

The values in the leftmost column show the elapsed time in milliseconds since Jan 1, 1970 *(known in programming circles as the epoch)* at the beginning of the run and at every 200th frame event thereafter.

The frame count and the frame rate

The values in the middle column show the frame count.

The values in the rightmost column show the frame rate computed from the amount of time required to display each set of 200 frames.

The default frame rate

This run was made without executing code to set the frame rate to a specific value. The values in the rightmost column in Figure 1 show the default frame rate for Flash Player 10 running on a Windows Vista machine. As you can see, the average frame rate is pretty solid at 30 frames per second.

Change the frame rate

Figure 2 shows the project output obtained by removing the comment markers and enabling the following statement in Listing 16.

stage.frameRate = 10;

The purpose of this statement is to set the frame rate for the Flash Player to ten frames per second.

Measured frame rate of Flash Player 10 at ten frames per second.

Figure

```
1271891012974
1271891033098 200 9.938382031405286
1271891053097 400 10.00050002500125
1271891073097 600 10
1271891093097 800 10
1271891113096 1000 10.00050002500125
```

Measured frame rate of Flash Player 10 at ten frames per second.

The average frame rate

As you can see in Figure 2, the average frame rate in this case is solid at ten frames per second.

A best-case scenario

This is probably the optimum case for measuring the frame rate since the program doesn't do anything else that may have an adverse impact on the ability of the Flash Player to maintain a constant frame rate.

Won't explain the code at this time

I'm not going to explain the code in Listing 16 at this time. You should have no difficulty understanding most of the code in Listing 16. Some of the code in Listing 16 is new to this lesson, but I will explain very similar code later in the lesson in conjunction with other projects.

Listen for TIMER events fired by a Timer object

Constructing a Timer object

The **flash.utils** package contains a class named **Timer** . An object instantiated from this class will fire a **TIMER** event every *(specified value)* milliseconds.

The constructor for the class requires two parameters:

1. The delay until the first event and between successive events in milliseconds as type **Number** .
2. The repeat count as type **int** , which specifies the number of events that will be fired. If this value if 0, the timer will fire events indefinitely.

The reciprocal of the rate

Note that whereas the frequency of frame events is specified as *frames per second* , the repetition parameter for timer events is specified as the time interval between events *(milliseconds per event)* . One is the reciprocal of the other.

You must start the timer

The timer does not start automatically. You must call the **start()** method to start it.

A project to measure the TIMER event rate

Listing 17 provides a simple AS3 project that illustrates the use of **TIMER** events. As before, this project computes and displays the average frequency of **TIMER** events for each of five consecutive sets of 200 events.

Two time bases running

In this case, there are two different time bases running: the frame rate and the timer rate. Unfortunately, it appears that they are not independent of one another.

A rate of 30 TIMER events per second

Figure 3 shows the output when the timer is set to fire 30 events per second and the Flash Player is running at its default rate of approximately 30 frames per second.
Measured Timer event rate at 30 events per second and 30 frames per second.

Figure

```
1271892152172
1271892159395 200 27.689325764917623
1271892166061 400 30.003000300030006
1271892172727 600 30.003000300030006
1271892179393 800 30.003000300030006
1271892186062 1000 29.9895036737142
```

Measured Timer event rate at 30 events per second and 30 frames per second.

Not too bad after a slow start

As you can see in Figure 3, after a somewhat slow start during the first 200 events, the timer event rate was reasonably solid at 30 events per second, on the average, for the next 800 events.

A rate of 10 TIMER events per second

Figure 4 shows the output when the timer was set to fire 10 events per second and the Flash Player was running at its default rate of approximately 30 frames per second.

Measured Timer event rate at 10 events per second and 30 frames per second.

Figure

```
1271892525653
1271892548767 200 8.65276455827637
1271892571894 400 8.64790072209971
1271892594995 600 8.657633868663694
1271892618445 800 8.528784648187633
1271892641413 1000 8.707767328456983
```

Measured Timer event rate at 10 events per second and 30 frames per second.

Not a very accurate event rate

In this case, the event rate was relatively *precise* at an average of about 8.65 events per second but didn't fare well in terms of *accuracy* since the target rate was 10 events per second.

Accuracy versus precision

Accuracy and precision don't mean the same thing. According to Wikipedia
,

> *"In the fields of engineering, industry and statistics, the accuracy of a measurement system is the degree of closeness of*

> *measurements of a quantity to its actual (true) value. The precision of a measurement system, also called reproducibility or repeatability, is the degree to which repeated measurements under unchanged conditions show the same results."*

Another result at a rate of 10 TIMER events per second

Figure 5 shows the output when the timer was set to produce 10 events per second and the Flash Player was running at a frame rate of 10 frames per second.
Measured Timer event rate at 10 events per second and 10 frames per second.

Figure

```
1271893113643
1271893134531 200 9.574875526618154
1271893155027 400 9.758001561280249
1271893175626 600 9.709209184911888
1271893195926 800 9.852216748768473
1271893216226 1000 9.852216748768473
```

Measured Timer event rate at 10 events per second and 10 frames per second.

Close, but no cigar

In this case, the event rate was closer to the target of 10 events per second, but was still consistently below the mark.

Even more troublesome is the difference between the results in Figure 4 and the results in Figure 5, which show that the event rate of the timer depends on the frame rate of the Flash Player.

An empty Flash Player window

The Flash Player was running in all five cases discussed above, but it was simply displaying an empty white window. No objects were being displayed in the Flash Player window.

A stand alone Flash Player

It is also worth noting that in all five cases, the standalone Flash Player was running, as opposed to the Flash Player plug-in for a browser. Therefore, these results were not influenced by the behavior of any specific browser.

Note: *The two projects discussed above were developed using the FlashDevelop tool. By default, the Test Movie option on the FlashDevelop Project Menu runs the project in a stand alone Flash Player and not in a browser plug in.*

Preview

In the remainder of this lesson, I will explain three different animation projects.

Run the ActionScript projects

If you have the Flash Player plug-in *(version 9 or later)* installed in your browser, you can click here to run the projects that I will explain in this lesson.

If you don't have the proper Flash Player installed, you should be notified of that fact and given an opportunity to download and install the Flash Player plug-in program.

Discussion and sample code

The project named Animation01

I will explain the remaining projects in this lesson in fragments. A complete listing of the code for the project named **Animation01** is provided in Listing 18 near the end of the lesson.

This is a very simple AS3 animation project. *(Note that it is not a Flex project.)*

I recommend that you run the online version of this project before continuing. *(See Deployment of FlashDevelop projects for information on how to deploy a release build from a FlashDevelop project on the Connexions website.)*

The ENTER_FRAME event stream

This project uses the **ENTER_FRAME** event stream as a time base to cause a filled blue circle drawn on a transparent **Sprite** object to move diagonally from left to right across the Flash window. The sprite with the circle moves out of the Flash window at the bottom right.

Beginning of the class named Main

This project was developed using the FlashDevelop tool. By default, the FlashDevelop tool names the required source code file and class definition **Main** . Listing 1 shows the beginning of the **Main** class for the project named **Animation01** .

Example:

Beginning of the class named Main.

```
package {
  import flash.display.Sprite;
  import flash.events.Event;

  public class Main extends Sprite{
    private var sprite:Sprite;
    private var dx:Number = 3;//x-movement
distance
    private var dy:Number = 2;//y-movement
distance
    private var radius:Number = 24;//radius of
circle
```

Not much that is new here

The class named **Main** extends the class named **Sprite** , which is a subclass of the class named **DisplayObject** several levels down the inheritance hierarchy. Therefore, the Flash Player will instantiate an object of this class and add it to the display list.

Otherwise, there is nothing in Listing 1 that should be new to you, so no further explanation of the code in Listing 1 should be required.

Beginning of the constructor

The beginning of the constructor for the **Main** class is shown in Listing 2.

Example:
Beginning of the constructor.

```
    public function Main():void {
      sprite = new Sprite();
      //Enable the following statement to cause 
the
      // sprite background to be visible.
      //sprite.opaqueBackground = true;

      //Draw a filled circle on the Sprite object;
      sprite.graphics.beginFill(0x0000ff, 1.0);

sprite.graphics.drawCircle(radius,radius,radius);
      sprite.graphics.endFill();

      addChild(sprite);//default location is 0,0
```

Another object of type Sprite

Listing 2 instantiates another object of the **Sprite** class and draws a blue filled circle on that object.

Then it adds that object to the display list as a child of the object of the class **Main** .

Draw a filled circle

According to the documentation, the **beginFill** method that is called on the **Sprite** object in Listing 2:

> *"Specifies a simple one-color fill that subsequent calls to other **Graphics** methods (such as **lineTo()** or **drawCircle()**) use when drawing. The fill remains in effect until you call the **beginFill()** , **beginBitmapFill()** , or **beginGradientFill()** method. Calling the **clear()** method clears the fill. The fill is not rendered until the **endFill()** method is called."*

That should be a sufficient explanation of how the code in Listing 2 draws a filled circle on the new **Sprite** object.

Add the new Sprite object to the display list

As mentioned earlier, the last statement in Listing 2 adds the new **Sprite** object to the display list as a child of the object of the **Main** class. **By default** , the new **Sprite** object is added at coordinates 0,0 which is the upper-left corner of the **Main** object.

Register an ENTER_FRAME listener

Listing 3 registers an event handler method named **onEnterFrame** that will be executed each time the Flash Player enters a new display frame.

Listing 3 also signals the end of the constructor.

Example:
Register an ENTER_FRAME listener.

```
      addEventListener(Event.ENTER_FRAME,
onEnterFrame);
    }//end constructor
```

The event handler method named onEnterFrame

The event handler method named **onEnterFrame** is shown in its entirety in Listing 4.

Example:
The event handler method named onEnterFrame.

```
    public function onEnterFrame(event:Event):void
{
      sprite.x += dx;
      sprite.y += dy;
    }//end onEnterFrame

  }//end class

}//end package
```

Properties named x and y

The **Sprite** object that was instantiated in Listing 2 has properties named **x** and **y** . The documentation has this to say about the property named **x** .

> *"Indicates the x coordinate of the DisplayObject instance relative to the local coordinates of the parent DisplayObjectContainer."*

The description of the property named **y** is very similar.

What this means in practice

What this means in practice is that the Flash Player uses the values of **x** and **y** properties as coordinates to decide where to draw the **Sprite** object relative to the upper-left corner of its container, which in this case is the object of the **Main** class.

By default, these property values are zero, which explains why the new **Sprite** object is initially drawn at the upper-left corner of the **Main** object.

Modify the x and y property values

Each time the Flash Player fires an **ENTER_FRAME** event, the code in Listing 4 increases the values of the **x** and **y** property values of the **Sprite** object by the amounts assigned to **dx** and **dy** in Listing 1.

Draw the sprite in a new location

The next time the objects in the display list are rendered on the screen, the **Sprite** object containing the filled blue circle will be drawn a little further to the right and a little further down the screen. That is what causes the blue ball to move diagonally from left to right across the Flash Player window when you run the project.

Nothing to stop it

Since there is nothing in the code to reverse the process of incrementing the **x** and **y** property values, the blue ball continues moving until it disappears off the Flash Player window on the lower-right side.

I will do something about that in the next project, which causes an image that is a caricature of me to bounce around inside of a rectangle.

The project named Animation07

A major upgrade

This project is a major upgrade of the project named **Animation01** in several respects. I recommend that you run this project before continuing.

Draw a rectangle

First, the constructor for the **Main** class draws a 450 x 500 pixel rectangle with a yellow background and border with a thickness of three pixels. *(The constructor begins in Listing 5 and continues into Listing 6.)*

An object of a new class

Then the constructor instantiates an object of a new **MoveableImage** class, passing the dimensions of the rectangle to the constructor for that class and adds that object to the display list.

An event handler

Finally, the constructor for the **Main** class registers an **ENTER_FRAME** event handler, which asks the **MoveableImage** object to move each time it is called. *(See Listing 7.)*

The MoveableImage class

The **MoveableImage** class extends the **Sprite** class and embeds an image in the **Sprite** object when it is instantiated. *(See Listing 8 and Listing 9.)*

The dimensions of a rectangle

The constructor for the **MoveableImage** class *(see Listing 10)* receives the dimensions of a rectangle as incoming parameters and saves those dimensions for later use.

The moveIt method

The **MoveableImage** class defines a method named **moveIt** *(see Listing 11)*
.

Each time the **moveIt** method is called, the object, *(including the embedded image)* moves by a prescribed distance in the horizontal and vertical directions.

Bounce off the edges

Whenever the object collides with an edge of the rectangle, it bounces off the edge and starts moving in a different direction.

Beginning of the Main class for the project named Animation07

As before, I will explain this program in fragments. A complete listing of the **Main** class for the program is provided in Listing 19.

The **Main** class begins in the fragment shown in Listing 5.

Example:
Beginning of the Main class for the project named Animation07.

```
package {
  import flash.display.Sprite;
  import flash.events.Event;

  public class Main extends Sprite{
    private var moveableImage:MoveableImage
    private var rectWidth:uint = 450;
    private var rectHeight:uint = 500;

    public function Main() {

      //Draw a black rectangle with a yellow
background.
      this.graphics.beginFill(0xFFFF00,1.0);
```

The only things that are new in Listing 5 are the two statements that call the **lineStyle** and **drawRect** methods of the **Graphics** class.

The first of the two statements sets the line style for the rectangle to be three pixels thick and to be black.

The second of the two statements sets the upper left corner of the rectangle to a coordinate position of 0,0 in the parent container and sets the width and height to the values established earlier when the width and height variables were declared and initialized.

Completion of the constructor for the Main class

Listing 6 completes the constructor for the **Main** class.

Example:
Completion of the constructor for the Main class.

```
      moveableImage = 
                  new 
MoveableImage(rectWidth,rectHeight);
      addChild(moveableImage);//default location 
is 0,0

      addEventListener(Event.ENTER_FRAME, 
onEnterFrame);
    }//end constructor
```

A new object of a custom class

Listing 6 begins by instantiating an object of the new custom class named **MoveableImage** , passing the width and height of the rectangle to the constructor for that class.

Then Listing 6 adds the new object to the display list. This will cause it to be rendered in the Flash Player window during the next display frame.

Finally, Listing 6 registers an event handler for **ENTER_FRAME** events.

The ENTER_FRAME event handler

The event handler is shown in Listing 7. Each time the **onEnterFrame** method is called, a message is sent to the **MoveableImage** object asking it to execute its **moveIt** method.

Example:
The ENTER_FRAME event handler.

```
    public function onEnterFrame(event:Event):void
{
      //Ask the image to move.
      moveableImage.moveIt();
    }//end onEnterFrame

  }//end class

}}//end package
```

The end of the Main class

Listing 7 also signals the end of the **Main** class.

Beginning of the MoveableImage class

A complete listing of the **MoveableImage** class is provided in Listing 20. The beginning of the **MoveableImage** class is shown in the code fragment in Listing 8.

Example:
Beginning of the MoveableImage class for the project named Animation07.

```
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.display.Bitmap;
  public class MoveableImage extends Sprite{

    private var dx:Number = 4;//x-movement 
distance
    private var dy:Number = 2;//y-movement 
distance
    private var rectWidth:uint;
    private var rectHeight:uint;
    private var imageWidth:uint;
    private var imageHeight:uint;
```

There is nothing new in Listing 8.

Embed the image in the swf file

The constructor for the **MoveableImage** class continues in Listing 9. The code in Listing 9 extracts an image from the specified image file and embeds it in the swf file with a reference named **headImage** .

Example:
Embed the image in the swf file.

```
    [Embed(source='/baldwin.jpg')]
    private var imgClass:Class;
    private var headImage:Bitmap = new imgClass
();
```

New to this lesson

I'm not going to try to explain how and why it works. I will simply suggest that you memorize the syntax for the next time that you need to do the same thing.

In addition, I will refer you to the following website where you will find an explanation.

The constructor for the MoveableImage class

The constructor for the class is shown in its entirety in Listing 10.

Example:
The constructor for the MoveableImage class.

```
    public function MoveableImage(rectWidth:uint,
                                  rectHeight:uint)
{
      //Save the dimensions of the rectangle.
      this.rectWidth = rectWidth;
      this.rectHeight = rectHeight;

      //Get and save the dimensions of the image.
```

The first two statements in Listing 10 simply save the width and height of the rectangle for later use. That shouldn't be new to you.

The next two statements use the reference to the embedded image *(**headImage**)* to get and save the width and height of the image referred to by the instance variable *(see Listing 9)* named **headImage** .

The last statement in Listing 10 adds that image to the display list as a child of the **Sprite** object.

The moveIt method of the MoveableImage class

The **moveIt** method is shown in its entirety in Listing 11.

Example:
The moveIt method of the MoveableImage class.

```
    //Cause the Sprite object to move and bounce
off of
    // the edges of the rectangle.
    public function moveIt():void {

      //Test for a collision with the left or
right edge
      // of the rectangle.
      if (((this.x + dx) > (rectWidth -
imageWidth)
                                       || (this.x
< 0))) {
        dx *= -1;//Reverse horizontal direction
      }//end if

      //Test for a collision with the top or the
bottom
      // of the rectangle.
      if (((this.y + dy) > (rectHeight -
imageHeight)
                                       || (this.y
< 0))) {
        dy *= -1;//Reverse vertical direction.
      }//end if

      //Make the move.
      this.x += dx;
      this.y += dy;
    }//end onEnterFrame

  }//end class

}}//end package
```

The **moveIt** method modifies the **x** and **y** property values of the **Sprite** object so that it will be rendered in a different location during the next display frame.

The logic that causes the object to bounce off the edges may take a few minutes for you to unravel. Otherwise, you should have no trouble understanding the code in Listing 11.

The end of the class

Listing 11 signals the end of the **MoveableImage** class.

The project named Animation01A

The code for this project is shown in its entirety in Listing 21.

This project is essentially the same as **Animation01** except that this project creates a **Timer** object and uses **TIMER** events in place of **ENTER_FRAME** events for timing.

If you run this project, you will see that just like **Animation01** , it causes a filled blue circle that is drawn on a transparent **Sprite** object to move diagonally from left to right across the Flash window. The sprite with the circle moves out of the Flash window at the bottom right.

Because of the similarity of this project to the **Animation01** project, I am only going to discuss the code that is significantly different between the two projects.

New import directives for the project named Animation01A

Listing 12 shows two new import directives that are required to make it possible to instantiate an object of the **Timer** class.

Example:
New import directives for the project named Animation01A.

```
  import flash.events.TimerEvent;
  import flash.utils.Timer;
```

Instantiate a new Timer object

Listing 13 shows the code that instantiates a new **Timer** object. This object will fire a **TIMER** event every 30 milliseconds and will continue to fire **TIMER events** indefinitely.

Example:
Instantiation of a new Timer object.

```
private var timer:Timer = new Timer(30);
```

Register a TIMER event handler and start the timer

The code in Listing 14 registers an event handler on the **Timer** object and then starts the timer running.

Example:
Register a TIMER event handler and start the timer.

```
      timer.addEventListener(TimerEvent.TIMER, 
onTimer);
      timer.start();
```

The TIMER event handler

The **TIMER** event handler is shown in Listing 15. Except for the method signature, this is essentially the same code that you saw in the **ENTER_FRAME** event handler in Listing 4.

Example:
The TIMER event handler.

```
    public function onTimer(event:TimerEvent):void 
{
      x += dx;
      y += dy;
    }//end onTimer
```

That's a wrap

I'm going to let that be it for this lesson, which has concentrated on time bases and animation for ActionScript 3 projects.

Run the projects

I encourage you to run these projects from the web. Then copy the code from Listing 16 through Listing 21. Use that code to create your own projects. Compile and run the projects. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete project listings

Complete listings of the projects discussed in this lesson are provided below.

Example:
The Main class for the project named TimeBase01.

```
/*Project TimeBase01

The purpose of this project is to experiment with the use
of Event.ENTER_FRAME as a time base. The project computes
and displays the average frame rate over five consecutive
sets of 200 frames. It also shows how to set the frame
rate to something other than the default value.

Must be run in debug mode to display the text data.
**************************************************
```

```
*******/
package {
  import flash.display.Sprite;
  import flash.events.Event;

  public class Main extends Sprite{
    private var sprite:Sprite;
    private var date:Date;
    private var countA:uint = 0;
    private var baseTime:Number = 0;
    private var currentTime:Number = 0;

    public function Main():void {
      sprite = new Sprite();
      addEventListener(Event.ENTER_FRAME,
onEnterFrame);
//      stage.frameRate = 10;
    }//end constructor
    //--------------------------------------------
------//

    //Event handler.
    public function onEnterFrame(event:Event):void
{
      currentTime = new Date().time;
      if (countA == 0) {
        baseTime = currentTime;
        trace(baseTime);
      }//end if

      if ((countA > 0) && (countA < 1001)
                                  && (countA % 200
== 0)){
        trace(currentTime + " " + countA + " "
               + 1000 / ((currentTime - baseTime)
/ 200));
        baseTime = currentTime;
      }//end if
      countA++;
    }//end onEnterFrame
  }//end class

}//end package
```

Example:
The Main class for the project named TimeBase02.

```
/*Project TimeBase02

The purpose of this program is to experiment with
the use
of a Timer object as a time base. The program
computes
and displays the average tick rate of the Timer
object
over five consecutive sets of 100 frames . It also
shows
how to set the tick rate to a specific value and
how to
specify the number of ticks.
**************************************************
*******/
package {
  import flash.display.Sprite;
  import flash.events.TimerEvent;
  import flash.utils.Timer;

  public class Main extends Sprite{
    private var sprite:Sprite;
    private var timer:Timer = new
Timer(1000/10,1001);
```

```
    private var date:Date;
    private var countB:uint = 0;
    private var baseTime:Number = 0;
    private var currentTime:Number = 0;

    public function Main():void {
      sprite = new Sprite();
      timer.addEventListener(TimerEvent.TIMER,
onTimer);
      timer.start();

//      stage.frameRate = 10;
    }//end constructor
    //--------------------------------------------------
------//

    //Event handler
    public function onTimer(event:TimerEvent):void
{
      currentTime = new Date().time;
      if (countB == 0) {
        baseTime = currentTime;
        trace(baseTime);
      }//end if

      if((countB > 0) && (countB < 1001) &&
                                     (countB % 200
== 0)){
        trace(currentTime + " " + countB + " "
               + 1000 / ((currentTime - baseTime)
/ 200));
        baseTime = currentTime;
      }//end if
      countB++;
    }//end onTimer

    //--------------------------------------------------
```

```
------//

  }//end class

}//end package
```

Example:
The Main class for the project named Animation01.

```
/*Project Animation01
Extremely simple animation project.
This is an AS3 project and not a Flex project.

Uses the ENTER_FRAME event for timing.
Causes a filled blue circle that is drawn on a
transparent Sprite object to move diagonally
from left to right across the Flash window.

The sprite with the circle moves out of the
Flash window at the bottom right.
**************************************************
*******/
pacpackage {
  import flash.display.Sprite;
  import flash.events.Event;

  public class Main extends Sprite{
    private var sprite:Sprite;
    private var dx:Number = 3;//x-movement
distance
    private var dy:Number = 2;//y-movement
distance
    private var radius:Number = 24;//radius of
circle
```

```
    public function Main():void {
      sprite = new Sprite();
      //Enable the following statement to cause
the
      // sprite background to be visible.
      //sprite.opaqueBackground = true;

      //Draw a filled circle on the Sprite object;
      sprite.graphics.beginFill(0x0000ff, 1.0);

sprite.graphics.drawCircle(radius,radius,radius);
      sprite.graphics.endFill();

      addChild(sprite);//default location is 0,0
      addEventListener(Event.ENTER_FRAME,
onEnterFrame);
    }//end constructor
    //----------------------------------------------
------//

    //Event handler.
    public function onEnterFrame(event:Event):void
{
      sprite.x += dx;
      sprite.y += dy;
    }//end onEnterFrame

  }//end class

}//end package
```

Example:
The Main class for the project named Animation07.

```
/*Project Animation07

Classical bouncing ball project written as an AS3
project.
However, in this case the ball is a Sprite object
with
an embedded image of Dick Baldwin's caricature.

Causes the image to bounce around inside of a 450
x 500
rectangle.
**************************************************
*******/
package {
  import flash.display.Sprite;
  import flash.events.Event;

  public class Main extends Sprite{
    private var moveableImage:MoveableImage
    private var rectWidth:uint = 450;
    private var rectHeight:uint = 500;

    public function Main() {

      //Draw a black rectangle with a yellow
background.
      this.graphics.beginFill(0xFFFF00,1.0);
      this.graphics.lineStyle(3, 0x000000);
      this.graphics.drawRect(0, 0, rectWidth,
rectHeight);
      this.graphics.endFill();

      moveableImage =
                  new
MoveableImage(rectWidth,rectHeight);
      addChild(moveableImage);//default location
is 0,0
```

```
      addEventListener(Event.ENTER_FRAME,
onEnterFrame);
    }//end constructor
    //--------------------------------------------------
------//

    //Event handler.
    public function onEnterFrame(event:Event):void
{
      //Ask the image to move.
      moveableImage.moveIt();
    }//end onEnterFrame

  }//end class

}//end package
```

Example:
The MoveableImage class for the project named Animation07.

```
/*Class MoveableImage
Constructs a Sprite with an embedded image that
can be
moved around inside of a rectangle for which the
dimensions are passed to the constructor.
**************************************************
*******/
package {
  import flash.display.Sprite;
  import flash.events.Event;
  import flash.display.Bitmap;

  public class MoveableImage extends Sprite{
    private var dx:Number = 4;//x-movement
distance
    private var dy:Number = 2;//y-movement
distance
    private var rectWidth:uint;
    private var rectHeight:uint;
    private var imageWidth:uint;
    private var imageHeight:uint;


    [Embed(source='/baldwin.jpg')]
    private var imgClass:Class;
    private var headImage:Bitmap = new imgClass
();


    public function MoveableImage(rectWidth:uint,
                                  rectHeight:uint)
{
      //Save the dimensions of the rectangle.
      this.rectWidth = rectWidth;
      this.rectHeight = rectHeight;

      //Get and save the dimensions of the image.
      imageWidth = headImage.width;
      imageHeight = headImage.height;

      //Add the image to the display list.
      addChild(headImage);

    }//end constructor
    //----------------------------------------------
------//

    //Cause the Sprite object to move and bounce
off of
```

```
    // the edges of the rectangle.
    public function moveIt():void {

      //Test for a collision with the left or
right edge
      // of the rectangle.
      if (((this.x + dx) > (rectWidth -
imageWidth)
                                       || (this.x
< 0))) {
        dx *= -1;//Reverse horizontal direction
      }//end if

      //Test for a collision with the top or the
bottom
      // of the rectangle.
      if (((this.y + dy) > (rectHeight -
imageHeight)
                                       || (this.y
< 0))) {
        dy *= -1;//Reverse vertical direction.
      }//end if

      //Make the move.
      this.x += dx;
      this.y += dy;

    }//end onEnterFrame

  }//end class

}//end package
```

Example:

The Main class for the project named Animation01A.

```
/*Project Animation01A
Extremely simple animation program.
This is an AS3 project and not a Flex project.

This program is essentially the same as
Animation01
except that this program creates a Timer object
and
uses TIMER events in place of ENTER_FRAME events
for
timing.

Causes a filled blue circle that is drawn on a
transparent Sprite object to move diagonally
from left to right across the Flash window.

The sprite with the circle moves out of the
Flash window at the bottom right.
**************************************************
*******/
package {
  import flash.display.Sprite;
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Animation Fundamentals

- Files:
    - ActionScript0150\ActionScript0150.htm
    - ActionScript0150\Connexions\ActionScriptXhtml0150.htm

Note: PDF disclaimer: Although the Connexions site makes it possible for you to download a PDF file for this module at no charge, and also makes it possible for you to purchase a pre-printed version of the PDF file, you should be aware that some of the HTML elements in this module may not translate well into PDF.

Note: Deployment of FlashDevelop projects on the Connexions website: As of the initial publication of this module, the Connexions website requires that all resources be located in the same folder as the file named **index.cnxml** . However, the output from a *release build* with FlashDevelop places a JavaScript file named **swfobject.js** in a folder named **js** that is a child of the folder that contains all of the other files, including the file named **index.html** . You can flatten this structure for deployment on the Connexions website by moving the file named **swfobject.js** into the folder that contains **index.html** , and then modifying one line of html code in the file named **index.html** to cause it to refer to the file named **swfobject.js** in the same folder as itself.

-end-

Sound in ActionScript
Learn to play sounds in ActionScript, both continuously and on call. Also learn how handle events fired by SoundChannel objects.

Note: Click Sound03 to run the ActionScript program from this lesson. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

Note: All references to ActionScript in this lesson are references to version 3.0 or later.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. The lesson titled **Using Flex 3 in a Flex 4 World** was added later to accommodate the release of Flash Builder 4. *(See Baldwin's Flex programming website .)* You should study those lessons before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

A sound in ActionScript is represented by an object of the class named **Sound** .

When the information encapsulated in a **Sound** object is *played* , that process is represented by an object of the class named **SoundChannel** .

According to the documentation :

> *"The Sound class lets you work with sound in an application. The Sound class lets you create a new Sound object, load and play an external MP3 file into that object, close the sound stream, and access data about the sound, such as information about the number of bytes in the stream and ID3 metadata. More detailed control of the sound is performed through the sound source -- the SoundChannel or Microphone object for the sound -- and through the properties in the SoundTransform class that control the output of the sound to the computer's speakers."*

The program that I will explain in this lesson makes use of events and methods of the **Sound** class and the **SoundChannel** class.

You will find a lot of interesting and useful information in the document titled **Basics of working with sound** on the Adobe website, including the following:

> *"Although there are various sound file formats used to encode digital audio, ActionScript 3.0 and Flash Player support sound files that are stored in the mp3 format. They cannot directly load or play sound files in other formats like WAV or AIFF."*

A variety of different sound file converter programs are available on the web that can be used to convert other sound file formats into mp3 format.

Preview

Run the ActionScript program named Sound03

If you have the Flash Player plug-in *(version 10 or later)* installed in your browser, click here to run the program named **Sound03** .

If you don't have the proper Flash Player installed, you should be notified of that fact and given an opportunity to download and install the Flash Player plug-in program.

Demonstrates the use of sound with ActionScript

This project is intended to demonstrate the use of sound with ActionScript. It also displays the image of a cloudy sky shown in Figure 1, but the only purpose of the image is to make it obvious when the program starts running. Sound03 image.

Figure

Sound03 image.

Stormy weather

The project is designed to give you the impression of the sounds that you might hear while sitting on your covered deck looking at the sky during a thunder storm.

Four stormy-weather sounds

The project plays the following sounds extracted from mp3 files:

- rain
- wind
- sizzle
- thunder

The **rain** sound is continuous.

The **wind** sound is played on startup and then occasionally thereafter on the basis of a random number generator.

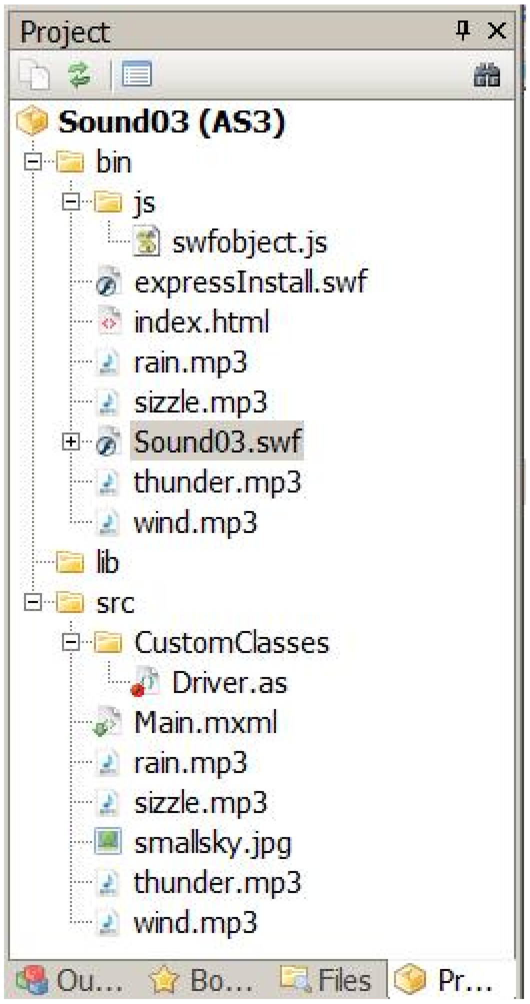
The **sizzle** sound is also played occasionally on the basis of a random number generator. *(You will probably need to be patient to hear this sound because it isn't played very often.)* As soon as the sizzle sound finishes, the sound of a **thunder** clap is played.

Discussion and sample code

Note: If you develop this project using FlashDevelop , you will need to manually copy all of the sound files into the bin folder.

The project file structure

The final project file structure, captured from the FlashDevelop project window, is shown in Figure 2.
Project file structure for Sound03.
Figure



Project file structure for Sound03.

As you can see in Figure 2, all of the sound and image files are stored in the folder named **src** . In addition, all of the sound files were manually copied into the folder named **bin** .

Will explain in fragments

I will explain the code for this program in fragments. Complete listings of the MXML code and the ActionScript code are provided in Listing 10 and Listing 11 near the end of the lesson.

The MXML code

The MXML code is shown in Listing 1 and also in Listing 10 for your convenience.

Example:
MXML code for the project named Sound03.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This project is intended to demonstrate the use of
sound
with ActionScript. See the file named Driver.as
for more
information
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

As is often the case in this series of tutorial lessons, the MXML file is very simple because the program was coded almost entirely in ActionScript. The MXML code simply instantiates an object of the **Driver** class. From that point forward, the behavior of the program is controlled by ActionScript code.

The ActionScript code

Import directives for the Driver class

The code for the **Driver** class begins in Listing 2, which shows the package declaration and the import directives.

Example:
Import directives for the Driver class

```
package CustomClasses{
  import flash.display.Bitmap;
  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.events.FlexEvent;
  import flash.events.TimerEvent;
  import flash.utils.Timer;
  import flash.media.Sound;
  import flash.net.URLRequest;
  import flash.media.SoundChannel;
  import flash.events.Event;
```

New to this lesson

The directives to import the **Sound** class, the **URLRequest** class, the **SoundChannel** class, and possibly the **Event** class are all new to this lesson.

Beginning of the Driver class proper

The definition of the **Driver** class begins in Listing 3.

Example:
Beginning of the Driver class proper.

```
  public class Driver extends Canvas {
    //Extending Canvas makes it possible to locate
    // images with absolute coordinates. The 
default
    // location is 0,0;

    private var smallSky:Image = new Image();

    //Instantiate a Timer object that will fire 
ten events
    // per second.
    private var timer:Timer = new Timer(100);

    //Declare a counter that will keep track of 
the number
    // of timer events that have been fired.
    private var loopCntr:uint;

    //Declare variables for the four sounds.
    private var sizzle:Sound;
    private var thunder:Sound;
    private var wind:Sound;
    private var rain:Sound;

    //Declare variables that are used to control 
when the
    // thunder sound is played.
    private var channel:SoundChannel;
    private var sizzlePlaying:Boolean = false;
```

Nothing new here

There is nothing new in Listing 3. I will call your attention to the declaration of variables of type **Sound** and **SoundChannel** . Otherwise, no explanation beyond the embedded comments should be required.

The constructor for the Driver class

The constructor for the **Driver** class is shown in its entirety in Listing 4.

Example:
The constructor for the Driver class.

```
    public function Driver(){//constructor
      //Load the sky image.
      //Note the use of a / to eliminate the
"Unable to
      // resolve asset for transcoding" Compiler
Error
      [Embed("/smallsky.jpg")]
      var imgSmall:Class;
      smallSky.load(imgSmall);

      //Load four sound files and play two of them
now.
      sizzle = new Sound();
      sizzle.load(new URLRequest("sizzle.mp3"));

      thunder = new Sound();
      thunder.load(new URLRequest("thunder.mp3"));

      wind = new Sound();
      wind.load(new URLRequest("wind.mp3"));
      //Play the wind sound through twice at
startup.
      wind.play(0,2);

      rain = new Sound();
      rain.load(new URLRequest("rain.mp3"));
      //Play the rain sound forever
      rain.play(0,int.MAX_VALUE);

      //Register an event listener on the 
CREATION_
      // COMPLETE event.

this.addEventListener(FlexEvent.CREATION_COMPLETE,

creationCompleteHandler);
    } //end constructor
```

There are quite a few things in Listing 4 that are new to this lesson.

Embed the image file

Although the code required to embed the image file in the swf file is not new to this lesson, it is worth highlighting the need to include the slash character to make the code compatible with the FlashDevelop IDE.

Load the sizzle sound

Listing 4 instantiates a new **Sound** object and stores the object's reference in the instance variable named **sizzle** . Then it calls the **load** method on that object to load the contents of the sound file named **sizzle.mp3** into the new **Sound** object.

The load method of the Sound class

Here is part of what the documentation has to say about the **load** method of the class named **Sound** :

> *"Initiates loading of an external MP3 file from the specified URL. If you provide a valid URLRequest object to the Sound*

> *constructor, the constructor calls Sound.load() for you. You only need to call Sound.load() yourself if you don't pass a valid URLRequest object to the Sound constructor or you pass a null value.*

> *Once load() is called on a Sound object, you can't later load a different sound file into that Sound object. To load a different sound file, create a new Sound object."*

Because I didn't provide a **URLRequest** object to the constructor when I instantiated the object of the **Sound** class, it was necessary for me to call the **load** method on the **Sound** object to load the sound file named **sizzle.mp3** .

Required parameter for the load method

Only one parameter is required by the **load** method and it must be of type **URLRequest** . To make a long story short, at least for the case where the sound file is located in the **src** folder as shown in Figure 2, you can create the required **URLRequest** object by calling the constructor for the **URLRequest** class and passing the name of the sound file as a **String** parameter to the constructor as shown in Listing 4.

Don't play the sizzle sound yet

The sizzle sound and the thunder sound are both encapsulated in **Sound** objects by the constructor in Listing 4. However, those sounds are not played by the constructor.

Encapsulate and play the wind wound

Listing 4 uses similar code to encapsulate the contents of the file named **wind.mp3** in an object of type **Sound** referred to by the instance variable named **wind** .

Then Listing 4 calls the **play** method on the **wind** object to cause the wind sound to be played from beginning to end twice when the program first starts running. *(It will be played again later at random times.)*

The play method of the Sound class

Here is part of what the documentation has to say about the **play** method of the **Sound** class:

> *"Generates a new SoundChannel object to play back the sound. This method returns a SoundChannel object, which you access to stop the sound and to monitor volume. (To control the volume, panning, and balance, access the SoundTransform object assigned to the sound channel.) "*

In other words, the play method causes the sound to start playing through a **SoundChannel** object, which you can manipulate to achieve various effects.

Didn't save a reference to SoundChannel object

Because I didn't have any need to manipulate the wind sound by way of the **SoundChannel** object, I didn't capture and save a reference to the object returned by the **play** method.

Parameters of the play method of the Sound class

The **play** method has three parameters, each of which has a default value. By default *(and this doesn't seem to agree with the documentation)* , if you call the **play** method on a **Sound** object and don't pass any parameters, the sound encapsulated in the object will be played once, starting at the beginning of the sound.

The first parameter

The first parameter is the initial position in milliseconds at which playback should start. The default value for this parameter is 0, which causes the sound to start at the beginning by default.

As is always the case with default parameters, if you want to provide a non-default value for the second parameter, you must also provide a value for the first parameter. When the **play** method is called on the **wind** sound in Listing 4, a value of 0 is passed as the first parameter to cause the sound to play from the beginning.

The second parameter

Instead of telling you what the documentation seems to say about the second parameter, I'm going to tell you how the second parameter behaves, which doesn't seem to agree with the documentation.

The value of the second parameter defines the number of times the sound will be played before the sound channel stops playback. For example, a value of 2 is passed as the second parameter when the **play** method is called on the **wind** sound in Listing 4. This parameter, in conjunction with the first parameter, causes the **wind** sound to be played through twice from the beginning to the end when the program starts running.

Rain, rain, go away: not any time soon

The maximum possible integer value is passed as the second parameter when the **play** method is called on the **rain** sound in Listing 4. This causes the rain sound to play over and over for a length of time that is probably longer than anyone would want to listen to it.

A CREATION_COMPLETE event listener

The last statement in the constructor in Listing 4 registers a **CREATION_COMPLETE** event handler on the **Canvas** object. You are already familiar with event listeners of this type. The code for the listener is shown in its entirety in Listing 5.

Example:
A CREATION_COMPLETE event handler.

```
    private function creationCompleteHandler(
                                  
event:mx.events.FlexEvent):void{
      
      //Set the width and height of the Canvas 
object
      // based on the size of the bitmap in the 
smallSky
      // image.
      this.width = Bitmap(smallSky.content).width;
      this.height = 
Bitmap(smallSky.content).height;
      
      //Add the image to this Canvas object.
      this.addChild(smallSky);
      
      //Register a timer listener and start the 
timer
      // running.
      timer.addEventListener(TimerEvent.TIMER, 
onTimer);
      timer.start();
      
    } //end creationCompleteHandler
```

As you learned in earlier lessons, this handler method is executed when the **Canvas** object has been fully created.

Nothing new here

There is nothing new in Listing 5. The code in Listing 5:

- Sets the width and the height of the **Canvas** object to match the width and the height of the image that is displayed while the program is running.
- Adds the image to the **Canvas** object.
- Registers an event listener on the **Timer** object that was instantiated in Listing 3 and starts the timer running to fire ten events per second.

Beginning of the TIMER event handler

The event handler that is registered on the **Timer** object begins in Listing 6. This method is executed each time the **Timer** object fires an event.

Example:
Beginning of the Timer event handler.

```
    public function onTimer(event:TimerEvent):void
{

      //Update the loop counter.
      loopCntr++;
      if (loopCntr == int.MAX_VALUE - 1) {
        //Guard against numeric overflow.
        loopCntr = 0;
      }//end if
```

Update the loop counter

Among other things, the code in the **Timer** event handler maintains a count of the number of events that have been fired by the timer. The code in Listing 6 increments the timer each time the event-handler method is executed, and sets the value back to zero when it reaches a very large value to guard against binary overflow.

Play an occasional wind sound

According to the code in Listing 3, the **Timer** object will fire an event every 100 milliseconds, or ten times per second. That causes the event handler to be called ten times per second.

The code in Listing 7 uses the modulus operator to identify every 25th call to the event handler. This occurs approximately once every 2.5 seconds, depending on the accuracy of the timer.

Example:
Play an occasional wind sound.

```
      if ((loopCntr % 25 == 0) && (Math.random() >
0.75)){
        wind.play();
      }//end if
```

Let the wind blow: or maybe not

When the code in Listing 7 determines that 2.5 seconds have passed since the last attempt to play the **wind** sound, it gets a random value of type **Number** with a value between 0 and 1.0. If that random value is greater than 0.75, it calls the **play** method on the **wind** sound to cause the sound to be played once from start to finish.

One wind sound every ten seconds on average

Assuming that the random values are uniformly distributed, about one out of every four random values will be greater than 0.75. Therefore, the wind sound should be played about once every ten seconds on average.

Play an occasional sizzle sound

Listing 8 uses a similar process to play an occasional sizzle sound. Listing 8 also causes the sizzle sound to be followed immediately by a clap of thunder.

Example:
Play an occasional sizzle sound.

```
      if ((loopCntr % 35 == 0) && (Math.random() > 
0.5)
                            && (sizzlePlaying == 
false)) {
        //Don't play another sizzle sound until 
this one
        // finishes.
        sizzlePlaying = true;

        //Play the sizzle sound and get a 
reference to the
        // SoundChannel object through which it is 
being
        // played.
        channel = sizzle.play();

        //Register an event listener that will be 
called
        // when the sizzle sound finishes playing.
        channel.addEventListener(
              Event.SOUND_COMPLETE, 
soundCompleteHandler);
      }//end if

    }//end onTimer
```

Save the SoundChannel reference

Listing 8 saves the **SoundChannel** reference returned by the **play** method in an instance variable named **channel** when the **play** method is called to play the **sizzle** sound.

Don't corrupt the reference to the SoundChannel object

In order to preclude the possibility of corrupting this reference by changing its value while the sound is playing, Listing 8 uses a **Boolean** instance variable named **sizzlePlaying** to guarantee that a new sizzle sound is not played before the previous one finishes.

The value of **sizzlePlaying** is set to true when the sizzle sound starts playing in Listing 8 and is set to false later when the sizzle sound finishes playing. Because the value of **sizzlePlaying** is tested by the conditional clause in the **if** statement in Listing 8, that conditional clause will never return true while **sizzlePlaying** is true.

Register a SOUND_COMPLETE event handler

The **SoundChannel** object fires a **SOUND_COMPLETE** event when the sound that it is playing finishes. Listing 8 registers an event listener on the **SoundChannel** object that is called each time the sizzle sound finishes playing. As you will see shortly, the code in the event handler sets the value of **sizzlePlaying** to false and also causes the thunder sound to be played as soon as the sizzle sound finishes.

The SOUND_COMPLETE event handler

The **SOUND_COMPLETE** event handler is shown in its entirety in Listing 9. This method is called each time the sizzle sound finishes playing.

Example:
The SOUND_COMPLETE event handler.

```
    private function
soundCompleteHandler(e:Event):void {
      //Allow another sizzle sound to be played
now that
      // this one is finished.
      sizzlePlaying = false;
      //Play the thunder immediately following the
end of
      // the sizzle sound.
      thunder.play();
    }//end soundCompleteHandler
    //--------------------------------------------
------//

  } //end class
} //end package
```

Allow another sizzle sound to be played

Listing 9 begins by setting the value of **sizzlePlaying** to false. This makes it possible for the sizzle sound to be played again when the other two expressions in the conditional clause of the **if** statement in Listing 8 return true.

Play a thunder clap

Then Listing 9 calls the **play** method on the **thunder** sound to cause the thunder sound to be played once immediately following the completion of each sizzle sound.

The end of the program

Listing 9 also signals the end of the **Driver** class and the end of the program.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 10 and Listing 11. Use that code to create your own project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the programs discussed in this lesson are provided below.

Example:
MXML code for the project named Sound03.

```
<?xml version="1.0" encoding="utf-8"?>
<!--
This project is intended to demonstrate the use of
sound
with ActionScript. See the file named Driver.as
for more
information
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">
  <cc:Driver/>

</mx:Application>
```

Example:
Class named Driver for the project named Sound03.

```
/*Project Sound03
This project is intended to demonstrate the use of
sound
with ActionScript. It also displays an image of a
cloudy
sky but the only purpose of the image is to make
it
obvious when the program starts running.

This project plays the following sounds extracted
from
mp3 files.

rain
wind
sizzle
thunder

The rain sound is continuous.

The wind sound is played occasionally on the basis
of a
random number generator.

The sizzle sound is also played occasionally on
the basis
of a random number generator. As soon as the
```

```
sizzle sound
finishes, a thunder clap sound is played.

Note that with FlashDevelop, you must manually put
a copy
of the sound files in the bin folder.
**************************************************
*******/
package CustomClasses{
  import flash.display.Bitmap;
  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.events.FlexEvent;
  import flash.events.TimerEvent;
  import flash.utils.Timer;
  import flash.media.Sound;
  import flash.net.URLRequest;
  import flash.media.SoundChannel;
  import flash.events.Event;


//==============================================
====//

  public class Driver extends Canvas {
    //Extending Canvas makes it possible to locate
    // images with absolute coordinates. The
default
    // location is 0,0;

    private var smallSky:Image = new Image();

    //Instantiate a Timer object that will fire
ten events
    // per second.
    private var timer:Timer = new Timer(100);
    //Declare a counter that will keep track of
the number
    // of timer events that have been fired.
    private var loopCntr:uint;

    //Declare variables for the four sounds.
    private var sizzle:Sound;
    private var thunder:Sound;
    private var wind:Sound;
    private var rain:Sound;

    //Declare variables that are used to control
when the
    // thunder sound is played.
    private var channel:SoundChannel;
    private var sizzlePlaying:Boolean = false;
    //--------------------------------------------
------//

    public function Driver(){//constructor
      //Load the sky image.
      //Note the use of a / to eliminate the
"Unable to
      // resolve asset for transcoding" Compiler
Error
      [Embed("/smallsky.jpg")]
      var imgSmall:Class;
      smallSky.load(imgSmall);

      //Load four sound files and play two of them
now.
      sizzle = new Sound();
      sizzle.load(new URLRequest("sizzle.mp3"));

      thunder = new Sound();
      thunder.load(new URLRequest("thunder.mp3"));
      wind = new Sound();
      wind.load(new URLRequest("wind.mp3"));
      //Play the wind sound through twice at
startup.
      wind.play(0,2);

      rain = new Sound();
      rain.load(new URLRequest("rain.mp3"));
      //Play the rain sound forever
      rain.play(0,int.MAX_VALUE);

      //Register an event listener on the
CREATION_
      // COMPLETE event.

this.addEventListener(FlexEvent.CREATION_COMPLETE,

creationCompleteHandler);
    } //end constructor
    //--------------------------------------------
------//

    //This handler method is executed when the
Canvas has
    // been fully created.
    private function creationCompleteHandler(

event:mx.events.FlexEvent):void{

      //Set the width and height of the Canvas
object
      // based on the size of the bitmap in the
smallSky
      // image.
      this.width = Bitmap(smallSky.content).width;
      this.height =
Bitmap(smallSky.content).height;
```

```
      //Add the image to this Canvas object.
      this.addChild(smallSky);

      //Register a timer listener and start the
timer
      // running.
      timer.addEventListener(TimerEvent.TIMER,
onTimer);
      timer.start();

    } //end creationCompleteHandler
    //-----------------------------------------
------//

    //TimerEvent handler. This method is executed
each
    // time the Timer object fires an event.
    public function onTimer(event:TimerEvent):void
{

      //Update the loop counter.
      loopCntr++;
      if (loopCntr == int.MAX_VALUE - 1) {
        //Guard against numeric overflow.
        loopCntr = 0;
      }//end if

      //Play an occasional wind sound.
      if ((loopCntr % 25 == 0) && (Math.random() >
0.75)){
        wind.play();
      }//end if

      //Play an occasional sizzle sound followed
      // immediately by a clap of thunder.
      if ((loopCntr % 35 == 0) && (Math.random() >
0.5)
                            && (sizzlePlaying ==
false)) {
        //Don't play another sizzle sound until
this one
        // finishes.
        sizzlePlaying = true;
        //Play the sizzle sound and get a
reference to the
        // SoundChannel object through which it is
being
        // played.
        channel = sizzle.play();
        //Register an event listener that will be
called
        // when the sizzle sound finishes playing.
        channel.addEventListener(
              Event.SOUND_COMPLETE,
soundCompleteHandler);
      }//end if

    }//end onTimer
    //--------------------------------------------
------//

    //This method is called each time the sizzle
sound
    // finishes playing. Each time it is called,
it plays
    // a thunder sound.
    private function
soundCompleteHandler(e:Event):void {
      //Allow another sizzle sound to be played
now that
      // this one is finished.
      sizzlePlaying = false;
      //Play the thunder immediately following the
```

```
end of
      // the sizzle sound.
      thunder.play();
    }//end soundCompleteHandler
    //--------------------------------------------------
------//

  } //end class
} //end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Sound in ActionScript
- Files:
    - ActionScript00160\ActionScript00160.htm
    - ActionScript00160\Connexions\ActionScriptXhtml00160.htm

-end-

Combining Sound with Motion and Image Animation
Learn to combine sounds, motion animation, image animation, and other interesting effects in a Flash movie using ActionScript 3.

Note: Click LighteningStorm01 to run the ActionScript program from this lesson. This program produces sound in addition to graphics. *(Click the "Back" button in your browser to return to this page.)*

Table of Contents

Preface

General

This tutorial lesson is part of a series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

The project that I will present and explain in this lesson is the culmination of several previous lessons dealing with animation, sound, transparency, mouse events, chroma key, etc.

Note: All references to ActionScript in this lesson are references to version 3.0 or later.

Several ways to create and launch ActionScript programs

There are several ways to create and launch programs written in the ActionScript programming language. Many of the lessons in this series will use Adobe Flex as the launch pad for the sample ActionScript programs.

An earlier lesson titled **The Default Application Container** provided information on how to get started programming with Adobe's Flex Builder 3. The lesson titled **Using Flex 3 in a Flex 4 World** was added later to accommodate the release of Flash Builder 4. *(See Baldwin's Flex programming website .)* You should study those lessons before embarking on the lessons in this series.

Some understanding of Flex MXML will be required

I also recommend that you study all of the lessons on Baldwin's Flex programming website in parallel with your study of these ActionScript lessons. Eventually you will probably need to understand both ActionScript and Flex and the relationships that exist between them in order to become a successful ActionScript programmer.

Will emphasize ActionScript code

It is often possible to use either ActionScript code or Flex MXML code to achieve the same result. Insofar as this series of lessons is concerned, the emphasis will be on ActionScript code even in those cases where Flex MXML code may be a suitable alternative.

Viewing tip

I recommend that you open another copy of this document in a separate browser window and use the following links to easily find and view the figures and listings while you are reading about them.

Figures

Listings

Supplemental material

I recommend that you also study the other lessons in my extensive collection of online programming tutorials. You will find a consolidated index at www.DickBaldwin.com .

General background information

Things you have learned

You learned about event handling, bitmap basics, the fundamentals of image pixel processing, and using chroma key compositing to create transparent backgrounds in earlier lessons in this series.

You also learned about drawing with the **Graphics** class, the fundamentals of animation and using sound in ActionScript in earlier lessons as well.

In the lesson titled **Animation Fundamentals** I told you that I refer to any program code that causes visual images to *change over time* to be animation.

Other types of animation

Although we commonly think of animation in terms of images that appear to move over time, there are many other valid forms of animation as well. For example, if the color of an image changes over time, that is animation. If the transparency of an image changes over time, that is animation. If graphic objects appear and then disappear, that is animation.

In this lesson, I will explain a program that is intended to pull together much of what you have already learned and to introduce you to this broader view of animation as well.

Preview

Run the ActionScript program named LighteningStorm01 .

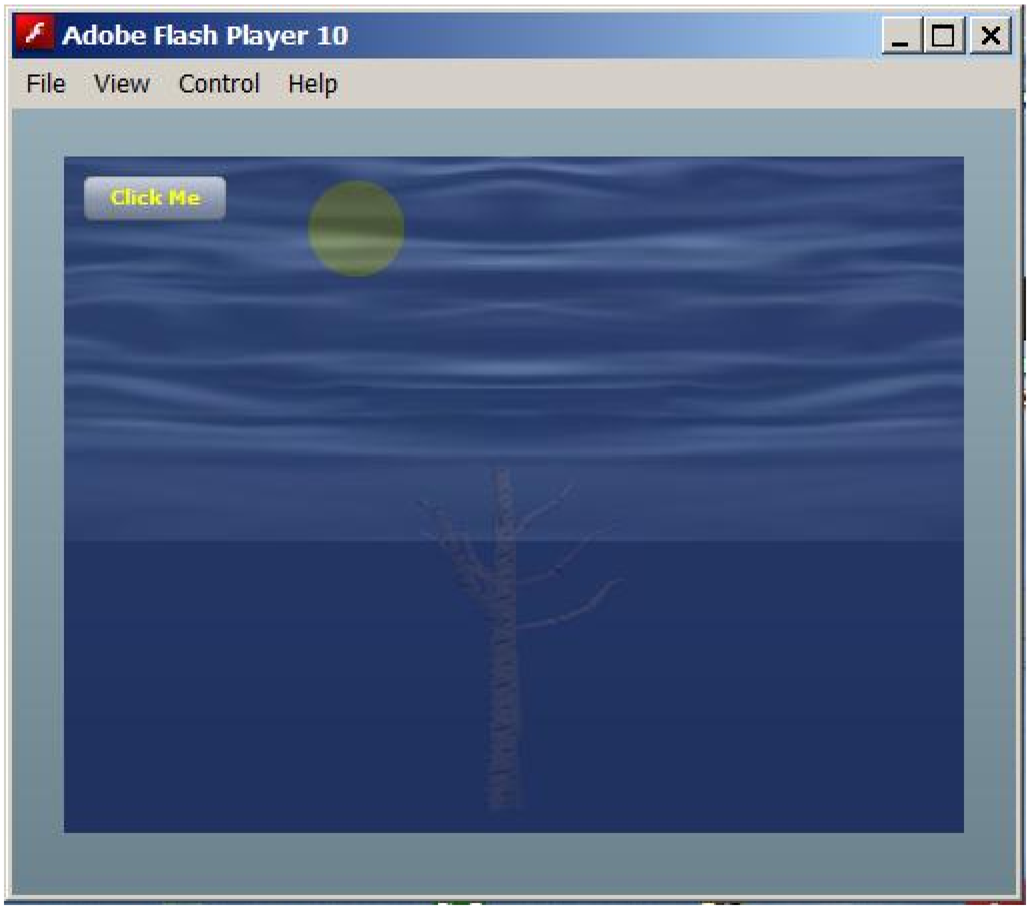
I recommend that you run the online version of this program before continuing.

Note: If you don't have the proper Flash Player installed, you should be notified of that fact and given an opportunity to download and install the Flash Player plug-in program.

LighteningStorm01 at startup

The program begins by displaying a scene similar to that shown in Figure 1.
LighteningStorm01 at startup.

Figure



LighteningStorm01 at startup.

What you should see

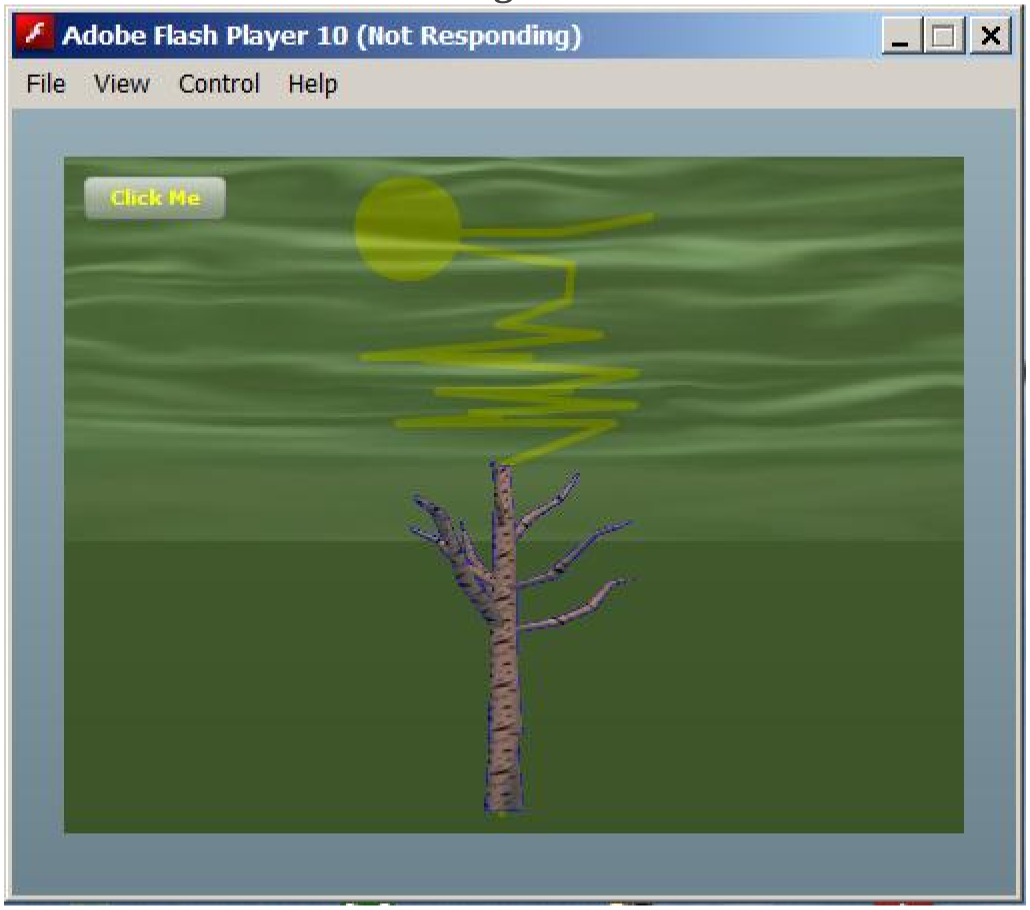
When the scene in Figure 1 appears, the clouds should seem to be moving slowly. The overall color of the scene should be changing slowly in the bluish-green range. The moon should be moving very slowly from left to right across the screen. You should be able to barely make out a tree in the fog near the bottom center of the image and you should be able to hear the wind and the rain. The sound of the rain should be continuous while the sound of the wind should come and go on a random basis.

Output produced by clicking the button

When you click the button, the scene should change to one similar to that shown in Figure 2.

Visual output produced by clicking the button.

Figure



Visual output produced by clicking the button.

The sights and sounds of a lightening bolt

You should hear a sizzle sound as a lightening bolt comes out of the sky and strikes the old tree. The shape of the lightening bolt should be random from one button-click to the next. Except for the motion of the moon and the shape of the tree, the details of all of the visual elements should change over time on in a random fashion.

The flash of the lightening

The lightening bolt should light up the scene with an eerie yellowish-green glow. The overall color of the scene should change slowly and randomly while the sizzle sound is playing and the lightening bolt is visible.

A loud clap of thunder

There should be a loud clap of thunder immediately following the sizzle sound as the scene reverts to something similar to that shown in Figure 1.

The moon

Throughout all of this, the moon should continue to move very slowly from left to right across the scene. When it reaches the right edge of the scene, it should wrap around and start over on the left side.

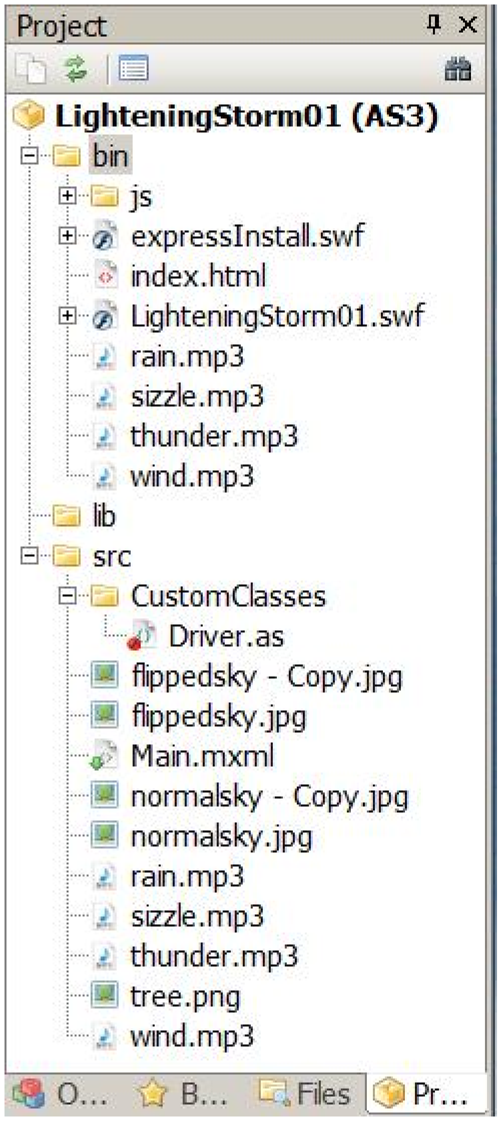
Discussion and sample code

The project file structure

The final project file structure, captured from the FlashDevelop project window, is shown in Figure 3.

Project file structure for LighteningStorm01.

Figure



Project

LighteningStorm01 (AS3)

```
bin
  js
  expressInstall.swf
  index.html
  LighteningStorm01.swf
  rain.mp3
  sizzle.mp3
  thunder.mp3
  wind.mp3
lib
src
  CustomClasses
    Driver.as
  flippedsky - Copy.jpg
  flippedsky.jpg
  Main.mxml
  normalsky - Copy.jpg
  normalsky.jpg
  rain.mp3
  sizzle.mp3
  thunder.mp3
  tree.png
  wind.mp3
```

Project file structure for LighteningStorm01.

As you can see in Figure 3, all of the sound and image files are stored in the folder named **src** . In addition, all of the sound files were manually copied into the folder named **bin** .

Will explain in fragments

I will explain the code for this program in fragments. Complete listings of the MXML code and the ActionScript code are provided in Listing 16 and Listing 17 near the end of the lesson.

The MXML code

The MXML code is shown in Listing 16. As is often the case in this series of lessons, the MXML file is very simple because the program was coded almost entirely in ActionScript. The MXML code simply instantiates an object of the **Driver** class. From that point forward, the behavior of the program is controlled by ActionScript code.

The ActionScript code

Beginning of the Driver class

The **Driver** class begins in Listing 1.

Example:
Beginning of the Driver class.

```
/*Project LighteningStorm01
This project was developed using FlashDevelop,
which
has a couple of requirements that may not exist
with
Flex Builder 3 or Flash Builder 4.
1. You must manually copy all mp3 files into the
bin
folder.
2. You must insert an extra slash character in the
URL
when embedding an image file in the swf file.
```

```
**************************************************
*******/
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.media.SoundChannel;
  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.controls.Button;
  import mx.events.FlexEvent;
  import flash.events.TimerEvent;
  import flash.events.MouseEvent;
  import flash.utils.Timer;
  import flash.utils.ByteArray;
  import flash.media.Sound;
  import flash.net.URLRequest;
  import flash.media.SoundChannel;
  import flash.events.Event;
  import flash.geom.Rectangle;



//==================================================
====//

  public class Driver extends Canvas {
    //Extending Canvas makes it possible to
position
    // images with absolute coordinates. The
default
    // location is 0,0;

    private var bkgndColor:uint = 0x005555;
    private var redBkgnd:uint = 0;
    private var greenBkgnd:uint = 128;
    private var blueBkgnd:uint = 128;
    private var normalSky:Image = new Image();
    private var flippedSky:Image = new Image();

    private var tree:Image = new Image();
    private var newTreeImage:Image = new Image();
    private var treeBitMap:Bitmap;

    private var alphaLim:Number = 0.5;
    private var normalAlpha:Number = alphaLim;
    private var flippedAlpha:Number;
    private var normalAlphaDecreasing:Boolean =
true;

    private var canvasObj:Canvas;
    private var timer:Timer = new Timer(35);
    private var loopCntr:uint;

    private var lighteningCntr:uint = 0;
    private var lighteningCntrLim:uint = 25;
    private var lighteningStartX:uint;
    private var lighteningStartY:uint;
    private var lighteningEndX:uint;
    private var lighteningEndY:uint;

    private var sizzle:Sound;
    private var thunder:Sound;
    private var wind:Sound;
    private var rain:Sound;

    private var sizzlePlaying:Boolean = false;
    private var channel:SoundChannel;

    private var button:Button;

    private var radius:Number = 24;//radius of
circle
    private var circleX:Number = 5 * radius;
    private var circleY:Number = 1.5 * radius;
    private var dx:Number = 0.05;
```

Nothing new here

There is nothing new in Listing 1, which consists almost entirely of import directives and instance variable declarations, so no further explanation of Listing 1 should be required. I will simply call your attention to the comments regarding the FlashDevelop IDE at the beginning of Listing 1.

Beginning of the constructor for the Driver class

The constructor for the **Driver** class begins in Listing 2.

Example:
Beginning of the constructor for the Driver class.

```
    public function Driver(){//constructor
      //Make this Canvas visible.
      bkgndColor = (redBkgnd << 16) + (greenBkgnd
<< 8)
                                              +
blueBkgnd;
      setStyle("backgroundColor", bkgndColor);
      setStyle("backgroundAlpha",0.5);
```

A 24-bit color value

The first statement in Listing 2 uses the left bitshift operator to construct a 24-bit binary value that will be used to establish the red, green, and blue values for the initial background color of the **Canvas** object. Hopefully you are already familiar with binary bit shifting. If not, just Google **bitshift operator** and you will find a lot of information on the topic. Note that the left bitshift operator is essentially the same in ActionScript, Java, C++, and other programming languages as well.

Set the initial background color and the transparency

Then Listing 2 calls the **setStyle** method twice in succession to set the background color and the transparency of that color for the background of the canvas.

An examination of the initial values for **redBkgnd** , **greenBkgnd** , and **blueBkgnd** in Listing 1 indicates that the initial background color is a dark shade of cyan with equal contributions of green and blue and no red.

The transparency of the background color

The second call to the **setStyle** method in Listing 2 causes the background color to exhibit a 50-percent transparency or opacity.

It is important that the background color not be completely opaque. If it were opaque, it would not be possible to see the yellow moon and the yellow lightening bolts that are drawn on the canvas behind the background color.

Will change over time

The background color will be changed later in an event handler that is registered on a **Timer** object. The color will not only be subject to small random changes. It will also be subject to major changes switching between the periods when a lightening flash is occurring or not occurring.

The remainder of the constructor

The remainder of the constructor is shown in Listing 3. There is nothing new in Listing 3 so no explanation beyond the embedded comments should be needed.

Example:
The remainder of the constructor.

```
      //Load the two sky images and embed them in
the
      // swf file.
      //Note the use of a / to eliminate the
"Unable to
      // resolve asset for transcoding" Compiler
Error
      [Embed("/normalsky.jpg")]
      var imgNormal:Class;
      normalSky.load(imgNormal);

      [Embed("/flippedsky.jpg")]
      var imgFlipped:Class;
      flippedSky.load(imgFlipped);

      //Load the treeImage and embed it in the swf
file.
      [Embed("/tree.png")]
      var imgTree:Class;
      tree.load(imgTree);


      //Load sound files and play two of them.
      sizzle = new Sound();
      sizzle.load(new URLRequest("sizzle.mp3"));

      thunder = new Sound();
      thunder.load(new URLRequest("thunder.mp3"));

      wind = new Sound();
      wind.load(new URLRequest("wind.mp3"));
      wind.play(0,2);//play twice

      rain = new Sound();
      rain.load(new URLRequest("rain.mp3"));
      rain.play(0, int.MAX_VALUE);//play forever


      //Register an event listener on the
CREATION_
      // COMPLETE event.

this.addEventListener(FlexEvent.CREATION_COMPLETE,

creationCompleteHandler);

      //Save a reference to this Canvas object,
which will
      // be used later for a variety of purposes.
      canvasObj = this;


      //Draw a yellow filled circle on this Canvas
object.
      graphics.beginFill(0xffff00);
      graphics.drawCircle(circleX,circleY,radius);
      graphics.endFill();

    } //end constructor
```

The CREATION_COMPLETE event handler

The code in the constructor in Listing 3 registers a **CREATION_COMPLETE** event handler on the **Canvas** object. You are already familiar with the use of event handlers of this type.

The **CREATION_COMPLETE** event handler is shown in its entirety in Listing 4. As before, there is nothing in Listing 4 that I haven't explained in previous lessons, so no explanation beyond the embedded comments should be needed.

Example:
The CREATION_COMPLETE event handler.

```
    //This handler method is executed when the
Canvas has
    // been fully created.
    private function creationCompleteHandler(

event:mx.events.FlexEvent):void{

      //Set the width and height of the Canvas
object
      // based on the size of the bitmap in the
      // normalSky image.
      this.width =
Bitmap(normalSky.content).width;
      this.height =
Bitmap(normalSky.content).height;

      //Add the images to the Canvas object. Note
that
      // the two images are overlaid at 0,0.
      this.addChild(normalSky);
      this.addChild(flippedSky);

      //Add a button in the upper-left corner in
front
      // of the sky images and register a CLICK
event
      // handler on the button.
      button = new Button();
      button.x = 10;
      button.y = 10;
      button.addEventListener(MouseEvent.CLICK,
onClick);
      button.label = "Click Me";
      button.setStyle("color", 0xFFFF00);
      addChild(button);


      //Get and save a reference to a Bitmap
object
      // containing the contents of the tree file.
      treeBitMap = Bitmap(tree.content);

      //Place the treeBitMap in a new Image object
and
      // place it on the canvas near the bottom
center of
      // the canvas.
      treeBitMap.x =
                 canvasObj.width / 2 -
treeBitMap.width/2;
      treeBitMap.y = canvasObj.height -
treeBitMap.height;

      newTreeImage.addChild(treeBitMap);
      this.addChild(newTreeImage);

      //Make the tree almost invisible. It will be
made
      // highly visible in conjunction with a
      // lightening flash.
      newTreeImage.alpha = 0.2;

      //Cause the blue background of the tree to
      // be transparent.
      processChromaKey(treeBitMap);
      //Register a timer listener and start the
timer
      // running.
      timer.addEventListener(TimerEvent.TIMER,
onTimer);
      timer.start();

    } //end creationCompleteHandler
```

Beginning of the TIMER event handler

The last two statements in Listing 4 register a **TIMER** event listener on the **Timer** object that was instantiated in Listing 1 and start the timer running.

The **TIMER** event handler begins in Listing 5. There is some interesting new code in this method, so I will break it down and explain it in fragments.

Example:
Beginning of the TIMER event handler.

```
    //TimerEvent handler. This method is executed
each
    // time the timer object fires an event.
    public function onTimer(event:TimerEvent):void
{

      //Update the loop counter. Several things
depend on
      // this counter.
      loopCntr++;
      if (loopCntr > int.MAX_VALUE-2) {
        //Guard against numeric overflow.
        loopCntr = 0;
      }//end if

      //Play a wind sound every 100th timer event
only
      // if a random value is greater than 0.5.
This
      // should happen half the time on the
average.
      if ((loopCntr % 100 == 0)&& (Math.random() >
0.5)) {
        wind.play();
      }//end if

      //Make random changes to the background
color.
      processBackgroundColor();

      //Make changes to the alpha values of the
normal
      // and flipped sky images.
      makeTheCloudsMove();
```

Approximately three Timer events per second

As you are aware, the event handler that begins in Listing 5 is executed each time the **Timer** object fires an event. The **Timer** object was instantiated in Listing 1 and configured to fire an event every 35 milliseconds, or approximately three times per second.

The only thing that is new in the fragment shown in Listing 5 is the pair of calls to the methods named:

- processBackgroundColor, and
- makeTheCloudsMove

The purpose of each of these methods is described by its name. I will explain both methods later in this lesson.

Controlling the motion of the moon

The remainder of the **Timer** event handler, which is shown in Listing 6, is dedicated to causing the moon to move very slowly from left to right across the screen as shown in Figure 1 and Figure 2.

Example:
Controlling the motion of the moon.

```
      //Draw a filled circle on this Canvas 
object.
      if (!sizzlePlaying) {
        //Erase the circle. Note that this would 
also
        // erase the lightening bolt if it were 
done while
        // the sizzle sound is playing.
        graphics.clear();
      }//end if

      //Make the circle move a very small distance 
to the
      // right. Make it wrap and reappear on the 
left
      //when it reaches the right side of the 
window.
      circleX += dx;
      if (circleX > canvasObj.width - radius) {
        circleX = 5 * radius;
      }//end if
      graphics.beginFill(0xffff00);
      graphics.drawCircle(circleX,circleY,radius);
      graphics.endFill();
```

```
    }//end onTimer
```

The code in Listing 6 draws a yellow filled circle a little further to the right each time the **Timer** fires an event. When the circle reaches the right edge of the Flash window, it starts over again on the left.

Erase the old moon before drawing the new moon

It is necessary to erase the old circle before drawing each new filled circle. Otherwise, instead of seeing a filled circle moving from left to right, the viewer would see a very wide yellow line being slowly drawn across the screen.

Houston, we have a problem

The proper way to erase the old filled circle is to call the **clear** method of the **Graphics** class. However, this is also the proper way to erase the yellow lightening bolt that I will explain later. Therefore, it is critical that the **clear** method not be called while the lightening bolt is on the screen.

The Boolean variable named sizzlePlaying

The **Boolean** variable named **sizzlePlaying** is used to control several aspects of the program relative to the period during which the sizzle sound is played, the lightening bolt is drawn, and the scene is illuminated by the lightening bolt.

The value of this variable is set to false when the variable is declared in Listing 1. It is set to true when the sizzle sound begins playing and is set back to false when the sizzle sound finishes playing. Thus, it is always true while the sizzle sound is playing and is false at all other times.

An egg-shaped moon

The value of **sizzlePlaying** is used in Listing 6 to prevent the **clear** method from being called while the lightening bolt is on the screen. This actually causes the moon to take on a slight egg shape during that period because new versions of the moon are being drawn without erasing the old versions. However, this isn't visually apparent because the moon moves such a short distance during that period. Also, the lightening bolt and not the moon probably commands the attention of the viewer during this period so the distortion isn't very noticeable.

Beyond that, no explanation of the code in Listing 6 beyond the embedded comments should be needed.

The method named processBackgroundColor

As you saw in Listing 5, the **Timer** event handler calls a method named **processBackgroundColor** each time the **Timer** object fires an event *(about three times per second)* . The purpose of the method is to cause the overall color of the image to change slowly over time. The method is shown in its entirety in Listing 7.

Example:
The method named processBackgroundColor.

```
    //This method processes the background color.
The
    // color changes among various shades of cyan
when
    // there is no lightening bolt. The color
changes
    // among various shades of dark yellow when
there is a
    // lightening bolt.
    private function processBackgroundColor():void
{
      if (!sizzlePlaying) {
        //Vary background color when there is no
        // lightening flash.
        if (Math.random() > 0.5) {
          if (greenBkgnd < 250){
            greenBkgnd += 5;
          }//end if
        }else {
          if(greenBkgnd > 5){
            greenBkgnd -= 5;
          }//end if
        }//end else

        if (Math.random() > 0.5) {
          if (blueBkgnd < 250){
            blueBkgnd += 5;
          }//end if
        }else {
          if(blueBkgnd > 5){
            blueBkgnd -= 5;
          }//end if
        }//end else

      }else {
        //Vary background color during a
lightening flash
        if (Math.random() > 0.5) {
          if (greenBkgnd < 245){
            greenBkgnd += 10;
          }//end if
        }else {
          if(greenBkgnd > 10){
            greenBkgnd -= 10;
          }//end if
        }//end else

        if (Math.random() > 0.5) {
          if (redBkgnd < 245){
            redBkgnd += 10;
          }//end if
        }else {
          if(redBkgnd > 10){
            redBkgnd -= 10;
          }//end if
        }//end else
      }//end else

      bkgndColor = (redBkgnd << 16) + (greenBkgnd
<< 8)
                                                     +
blueBkgnd;
      setStyle("backgroundColor", bkgndColor);
      setStyle("backgroundAlpha",0.5);

    }//end processBackgroundColor
```

Long and tedious

The code in Listing 7 is long and tedious but not particularly complicated.

Three sections of code

The code can be broken down into three sections for purposes of explanation. The first section begins at the beginning of the **if** statement and continues down to, but not including the **else** clause. Note that the conditional clause for the **if** statement tests to determine if the value of the variable named **sizzlePlaying** is false.

For a value of false, the code in the **if** statement makes very small random changes to the green and blue components of the background color during those periods when there is no lightening bolt on the screen. The value of the red color component is zero during this period.

The second section

The second section begins with the **else** clause, and the code in this section is executed when the value of **sizzlePlaying** is true.

The code in this section makes very small random changes to the red and green components of the background color during those periods where there is a lightening bolt on the screen. The value of the blue color component is zero during this period.

The third section -- apply the color components

The third section of code, consisting of the last three statements, uses the red, green, and blue color component values computed earlier to cause the color of the background to change. Note that this code maintains a 50-percent opacity value for the background color.

The method named makeTheCloudsMove

As you also saw in Listing 5, the **Timer** event handler also calls a method named **makeTheCloudsMove** each time the **Timer** object fires an event, or about three times per second. The purpose of this method is to create the illusion that the clouds shown in Figure 1 and Figure 2 are moving.

The one new thing

The procedure for accomplishing this is probably the only thing in this lesson that I haven explained in one form or another in an earlier lesson.

The image of the clouds shown in Figure 1 and Figure 2 is actually the superposition of two images, one in front of the other. The two images are shown in Figure 4 and Figure 5.

The image named normalsky.jpg.

Figure

The image named normalsky.jpg.

The image named flippedsky.jpg.

Figure

The image named flippedsky.jpg.

The differences between the images

If you examine these two images carefully, you will see that:

- One is the mirror image of the other.
- One has been given a green tint while the other has been given a magenta tint.

As mentioned earlier, the program displays both of these images, one on top of the other.

The illusion of movement...

The illusion of movement is achieved by causing the alpha transparency value of one image to go down while the alpha transparency value of the other image goes up and vice versa. In other words, the two images are caused to gradually fade in and out in opposition to one another.

Beginning of the method named makeTheCloudsMove

The code that accomplishes this begins in Listing 8.

Example:
Beginning of the method named makeTheCloudsMove.

```
    //This method processes the alpha values of
the
    // normal and flipped sky images.
    // The change in alpha values of the
overlapping
    // images makes it appear that the clouds are
    // moving.
    private function makeTheCloudsMove():void {

      //Change the decreasing or increasing
direction of
      // the changes in the alpha value for the
normal
      // sky image when the alpha value hits the
limits.
      if (normalAlphaDecreasing && (normalAlpha <=
0.1)) {
        normalAlphaDecreasing = false;
      }else if (!normalAlphaDecreasing &&
                              (normalAlpha >=
alphaLim)) {
        normalAlphaDecreasing = true;
      }//end if
```

A saw tooth change in the alpha values

The alpha value for the normal sky image is caused to range from 0.1 to 0.5 in increments of 0.005 in a saw tooth fashion. At the same time, the alpha value for the other image is caused to range between the same limits in an opposing saw tooth fashion.

The code in Listing 8 keeps track whether the alpha values for the normal sky image are going up or going down, and flips the direction whenever the current alpha value crosses one of the limits.

Compute new alpha value for the normal sky

Listing 9 uses that information to compute a new alpha value for the normal sky image.

Example:
Compute new alpha value for the normal sky image.

```
      //Increase or decrease the alpha value for
the
      // normal sky image.
      if (normalAlphaDecreasing) {
        normalAlpha -= 0.005;
      }else {
        normalAlpha += 0.005;
      }//end else
```

Compute new alpha value for the flipped sky image

Listing 10 uses the new alpha value for the normal sky along with the upper limit of the alpha value to compute a new alpha value for the flipped sky. As the alpha value for the normal sky goes up, the alpha value for the flipped sky goes down and vice versa.

Example:
Compute new alpha value for the flipped sky image.

```
      //Cause the alpha value for the flipped sky
image
      // to go down when the value for the normal
sky
      // image goes up, and vice versa.
      flippedAlpha = alphaLim - normalAlpha;
```

Apply the new alpha values to both sky images

Finally, Listing 10 sets the alpha value for each image to the new value.

Example:
Apply the new alpha values to both sky images.

```
      //Change the alpha values for both sky
images.
      normalSky.alpha = normalAlpha;
      flippedSky.alpha = flippedAlpha;
    }//end makeTheCloudsMove
```

The next time the images are rendered, the new alpha values will be in effect.

The CLICK event handler for the button

The button shown in Figure 1 provides the mechanism by which the viewer can interact with the program.

The code in Listing 4 registers a **CLICK** event handler on the button. Listing 12 shows that **CLICK** event handler. This method is called each time the user clicks the button.

Example:
The CLICK event handler for the button.

```
    //This method is a click handler on the
button. It
    // causes the lightening flash to occur and
the
```

```
    // lightening bolt to be drawn.
    private function
onClick(event:MouseEvent):void {
      //Don't create another lightening bolt while
the
      // previous one is still in progress.
      if(!sizzlePlaying){
        flashLightening();
        drawLightening();
      }//end if
    }//end onClick
```

Create the lightening bolt and its flash

The code in Listing 12 first confirms that the sizzle sound is not currently being played. If not, Listing 12 calls the method named **flashLightening** to illuminate the scene, and calls the method named **drawLightening** to draw the lightening bolt.

The method named flashLightening

The method named **flashLightening** is shown in its entirety in Listing 13.

Example:
The method named flashLightening.

```
    private function flashLightening():void {

      //Make the tree more visible. Apparently
      // setting the alpha property has no effect
on the
      // alpha byte values that have been
individually
```

```
      // set. Otherwise, the blue background would
      // become visible.
      newTreeImage.alpha = 1.0;


      //Play a sizzle sound to accompany the flash
of
      // lightening. Set a flag to prevent another
sizzle
      // sound from being played before this one
finishes.
      sizzlePlaying = true;
      channel = sizzle.play();
      //Register an event listener that will be
called
      // when the sizzle sound finishes playing.
      channel.addEventListener(
              Event.SOUND_COMPLETE,
soundCompleteHandler);


      //Change the background color to a dark
yellow.
      redBkgnd = 128;
      greenBkgnd = 128;
      blueBkgnd = 0;

    }//end flashLightening
```

Produce the visual and audible effects of the lightening

The purpose of this method is to produce the visible and audible effects of the lightening other than the lightening bolt itself.

The method creates the flash from thelightening bolt, makes the tree more visible during the flash as shown in Figure 2, and plays a sizzle sound that will be followed by a clap of thunder.

Several steps are involved

The method begins by setting the alpha value on the tree image to 1.0 to cause the tree to become totally opaque.

Then it sets the value of the variable named **sizzlePlaying** to true to notify all other parts of the program that a sizzle sound is being played and a lightening bolt is being drawn.

Then it calls the **play** method on the **sizzle Sound** . The **play** method starts the sizzle sound playing and immediately returns a reference to a **SoundChannel** object through which the sound will be played.

The reference to the **SoundChannel** object is saved in the instance variable named **channel** .

Listing 13 registers an event listener on the **SoundChannel** object that will be executed when the sizzle sound finishes playing.

Finally, Listing 13 sets the red, green, and blue background color component values to dark yellow. These values along with the true value of **sizzlePlaying** will be used by the code in Listing 7 to set the background color of the canvas to dark yellow the next time the **Timer** object fires an event.

The drawLightening method

Listing 14 shows the method named **drawLightening** that is called by the **CLICK** event handler on the button in Listing 12 to draw the actual lightening bolt.

Example:
The method named drawLightening.

```
    private function drawLightening():void {

      lighteningStartX = Math.floor(Math.random()
                                    *
canvasObj.width / 3)
                                    +
canvasObj.width / 3;
      lighteningStartY =
                    Math.random() *
canvasObj.height / 10;
      lighteningEndX = canvasObj.width / 2 -6;
      lighteningEndY =
                canvasObj.height -
treeBitMap.height + 10;

      //Draw a zero width dark yellow line to the
starting
      // point of the lightening bolt.
      canvasObj.graphics.lineStyle(0, 0x999900);
      canvasObj.graphics.lineTo(
                      lighteningStartX,
lighteningStartY);

      //Set the line style to a bright yellow line
that is
      // four pixels thick.
      canvasObj.graphics.lineStyle(4, 0xFFFF00);

      //Declare working variables.
      var tempX:uint;
      var tempY:uint = lighteningStartY;
      var cnt:uint;

      //Use a for loop to draw a lightening bolt
with
      // twenty random segments.
      for (cnt = 0; cnt < 20; cnt++ ) {
        //Compute the coordinates of the end of
the next
        // line segment.
        tempX = Math.floor(Math.random()
                                    *
canvasObj.width / 3)
                                    +
canvasObj.width / 3;
        tempY = tempY + Math.floor(Math.random()
                            * (lighteningEndY -
tempY)/5);
        //Draw the line segment.
        canvasObj.graphics.lineTo(tempX,tempY);
      }//end for loop

      //Draw a line segment to the top of the
tree.
      canvasObj.graphics.lineTo(
                          lighteningEndX,
lighteningEndY);

      //Make the lightening go to ground.
      canvasObj.graphics.lineTo(
          lighteningEndX,
             lighteningEndY + treeBitMap.height -
20);
    }//end drawLightening
```

This method draws a yellow segmented lightening bolt four pixels thick *(as shown in Figure 2)* that is generally random but always ends up striking the top of the tree.

Long and tedious

As was the case earlier, this method is long and tedious but not technically difficult. Therefore, I will leave it as an exercise for the student to wade through the details in order to understand how it draws the lightening bolt.

The method named soundCompleteHandler

That brings us to the method shown in Listing 15 that is called each time a sizzle sound finishes playing.

Example:
The method named soundCompleteHandler.

```
    private function
soundCompleteHandler(e:Event):void {

      //Allow another sizzle sound to be played
now that
      // this one is finished.
      sizzlePlaying = false;
      //Play the thunder immediately following the
end of
      // the sizzle sound.
      thunder.play();

      //Switch the background color from dark
yellow
      // to the normal background color.
      redBkgnd = 0;
      greenBkgnd = 128;
      blueBkgnd = 128;

      //Erase the lightening bolt. Note that this
will
      // also erase the yellow circle.
      canvasObj.graphics.clear();
      //Make the tree almost invisible.
      newTreeImage.alpha = 0.2;

    }//end soundCompleteHandler
```

Each time this method is called, it sets the **sizzlePlaying** variable to false to clear the way for the sizzle sound to be played again.

Then it plays the thunder sound and sets the color variables so that the background color will be restored to a dark cyan color by the code in Listing 7.

Finally it calls the **clear** method of the **Graphics** class to erase the lightening bolt, which also erases the moon as well.

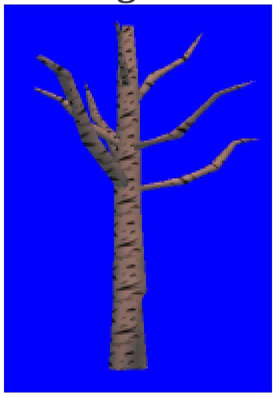
Then it sets the alpha value on the tree image to a low value to make the tree appear to be lost in the fog.

The method named processChromaKey

That leaves only the method named **processChromaKey** that I haven't explained. The purpose of this method is to cause the blue background pixels of the tree image shown in Figure 6 to become transparent.

The tree image.

Figure



The tree image.

This method is essentially the same as a method that I explained in an earlier lesson titled **Using Chroma Key Compositing to Create Transparent Backgrounds** . Rather than to explain that method again, I will simply refer you to the earlier lesson for an explanation. You can view the method in its entirety in Listing 17.

Run the program

I encourage you to run this program from the web. Then copy the code from Listing 16 and Listing 17. Use that code to create your own project. Compile and run the project. Experiment with the code, making changes, and observing the results of your changes. Make certain that you can explain why your changes behave as they do.

Resources

I will publish a list containing links to ActionScript resources as a separate document. Search for ActionScript Resources in the Connexions search box.

Complete program listings

Complete listings of the programs discussed in this lesson are provided below.

Example:
Code for Main.mxml

```
<?xml version="1.0" encoding="utf-8"?>
<!--
Project LighteningStorm01
```

```
See Driver.as for a description of this project.
-->

<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:cc="CustomClasses.*">

  <cc:Driver/>

</mx:Application>
```

Example:
Source code for the class named Driver.

```
/*Project LighteningStorm01
This project is the culmination of several
previous
projects dealing with animation, sound,
transparency,
mouse events, chromakey, etc.

When the program starts running, the scene is of a
very
stormy day. There is a button in the upper-left
corner
of the scene. The clouds are moving. There is also
a
yellow moon behind the clouds that is moving very
slowly across the screen from left to right.

When the user clicks the button, a bolt of
lightening
comes out of the sky and strikes an image of a
tree.
```

```
Several aspects of the scene change to reflect
the sights and sounds of a lightening strike.

In addition to the clouds moving, the overall
color
of the scene slowly changes randomly. The overall
color varies around a dark cyan when there is no
lightening bolt and varies around a dark yellow
when
there is a lightening bolt.

This project was developed using FlashDevelop,
which
has a couple of requirements that may not exist
with
Flex Builder 3 or Flash Builder 4.
1. You must manually copy all mp3 files into the
bin
folder.
2. You must insert an extra slash character in the
URL
when embedding an image file in the swf file.
**************************************************
*******/
package CustomClasses{
  import flash.display.Bitmap;
  import flash.display.BitmapData;
  import flash.media.SoundChannel;
  import mx.containers.Canvas;
  import mx.controls.Image;
  import mx.controls.Button;
  import mx.events.FlexEvent;
  import flash.events.TimerEvent;
  import flash.events.MouseEvent;
  import flash.utils.Timer;
  import flash.utils.ByteArray;
  import flash.media.Sound;
  import flash.net.URLRequest;
  import flash.media.SoundChannel;
  import flash.events.Event;
  import flash.geom.Rectangle;



//==================================================
====//

  public class Driver extends Canvas {
    //Extending Canvas makes it possible to
position
    // images with absolute coordinates. The
default
    // location is 0,0;

    private var bkgndColor:uint = 0x005555;
    private var redBkgnd:uint = 0;
    private var greenBkgnd:uint = 128;
    private var blueBkgnd:uint = 128;

    private var normalSky:Image = new Image();
    private var flippedSky:Image = new Image();

    private var tree:Image = new Image();
    private var newTreeImage:Image = new Image();
    private var treeBitMap:Bitmap;

    private var alphaLim:Number = 0.5;
    private var normalAlpha:Number = alphaLim;
    private var flippedAlpha:Number;
    private var normalAlphaDecreasing:Boolean =
true;

    private var canvasObj:Canvas;
    private var timer:Timer = new Timer(35);
    private var loopCntr:uint;

    private var lighteningCntr:uint = 0;
    private var lighteningCntrLim:uint = 25;
    private var lighteningStartX:uint;
    private var lighteningStartY:uint;
    private var lighteningEndX:uint;
    private var lighteningEndY:uint;

    private var sizzle:Sound;
    private var thunder:Sound;
    private var wind:Sound;
    private var rain:Sound;

    private var sizzlePlaying:Boolean = false;
    private var channel:SoundChannel;

    private var button:Button;

    private var radius:Number = 24;//radius of
circle
    private var circleX:Number = 5 * radius;
    private var circleY:Number = 1.5 * radius;
    private var dx:Number = 0.05;
    //----------------------------------------
------//


    public function Driver(){//constructor
      //Make this Canvas visible.
      bkgndColor = (redBkgnd << 16) + (greenBkgnd
<< 8)
                                              +
blueBkgnd;
      setStyle("backgroundColor", bkgndColor);
      setStyle("backgroundAlpha",0.5);

      //Load the two sky images and embed them in
the
      // swf file.
      //Note the use of a / to eliminate the
"Unable to
      // resolve asset for transcoding" Compiler
Error
      [Embed("/normalsky.jpg")]
      var imgNormal:Class;
      normalSky.load(imgNormal);

      [Embed("/flippedsky.jpg")]
      var imgFlipped:Class;
      flippedSky.load(imgFlipped);

      //Load the . treeImage and embed it in the
swf file.
      [Embed("/tree.png")]
      var imgTree:Class;
      tree.load(imgTree);


      //Load sound files and play two of them.
      sizzle = new Sound();
      sizzle.load(new URLRequest("sizzle.mp3"));

      thunder = new Sound();
      thunder.load(new URLRequest("thunder.mp3"));

      wind = new Sound();
      wind.load(new URLRequest("wind.mp3"));
      wind.play(0,2);//play twice

      rain = new Sound();
      rain.load(new URLRequest("rain.mp3"));
      rain.play(0, int.MAX_VALUE);//play forever
      //Register an event listener on the
CREATION_
      // COMPLETE event.

this.addEventListener(FlexEvent.CREATION_COMPLETE,

creationCompleteHandler);

      //Save a reference to this Canvas object,
which will
      // be used later for a variety of purposes.
      canvasObj = this;


      //Draw a yellow filled circle on this Canvas
object.
      graphics.beginFill(0xffff00);
      graphics.drawCircle(circleX,circleY,radius);
      graphics.endFill();

    } //end constructor
    //--------------------------------------------
------//

    //This handler method is executed when the
Canvas has
    // been fully created.
    private function creationCompleteHandler(

event:mx.events.FlexEvent):void{

      //Set the width and height of the Canvas
object
      // based on the size of the bitmap in the
      // normalSky image.
      this.width =
Bitmap(normalSky.content).width;
      this.height =
Bitmap(normalSky.content).height;

      //Add the images to the Canvas object. Note
that
      // the two images are overlaid at 0,0.
      this.addChild(normalSky);
      this.addChild(flippedSky);

      //Add a button at in the upper-left corner
in front
      // of the sky images.
      button = new Button();
      button.x = 10;
      button.y = 10;
      button.addEventListener(MouseEvent.CLICK,
onClick);
      button.label = "Click Me";
      button.setStyle("color", 0xFFFF00);
      addChild(button);


      //Get and save a reference to a Bitmap
object
      // containing the contents of the tree file.
      treeBitMap = Bitmap(tree.content);

      //Place the treeBitMap in a new Image object
and
      // place it on the canvas near the bottom
center of
      // the canvas.
      treeBitMap.x =
                 canvasObj.width / 2 -
treeBitMap.width/2;
      treeBitMap.y = canvasObj.height -
treeBitMap.height;

      newTreeImage.addChild(treeBitMap);
      this.addChild(newTreeImage);

      //Make the tree almost invisible. It will be
made
      // highly visible in conjunction with a
      // lightening flash.
      newTreeImage.alpha = 0.2;

      //Cause the blue background of the tree to
      // be transparent.
      processChromaKey(treeBitMap);


      //Register a timer listener and start the
timer
      // running.
      timer.addEventListener(TimerEvent.TIMER,
onTimer);
      timer.start();

    } //end creationCompleteHandler
    //--------------------------------------------------
------//

    //TimerEvent handler. This method is executed
each
    // time the timer object fires an event.
    public function onTimer(event:TimerEvent):void
{

      //Update the loop counter. Several things
depend on
      // this counter.
      loopCntr++;
      if (loopCntr > int.MAX_VALUE-2) {
        //Guard against numeric overflow.
        loopCntr = 0;
      }//end if

      //Play a wind sound every 100th timer event
only
      // if a random value is greater than 0.5.
This
      // should happen half the time on the
average.
      if ((loopCntr % 100 == 0)&& (Math.random() >
0.5)) {
        wind.play();
      }//end if

      //Make random changes to the background
color.
      processBackgroundColor();

      //Make changes to the alpha values of the
normal
      // and flipped sky images.
      makeTheCloudsMove();

      //Draw a filled circle on this Canvas
object.
      if (!sizzlePlaying) {
        //Erase the circle. Note that this would
also
        // erase the lightening bolt if it were
done while
        // the sizzle sound is playing.
        graphics.clear();
      }//end if

      //Make the circle move a very small distance
```

```
to the
      // right. Make it wrap and reappear on the
left
      //when it reaches the right side of the
window.
      circleX += dx;
      if (circleX > canvasObj.width - radius) {
        circleX = 5 * radius;
      }//end if
      graphics.beginFill(0xffff00);
      graphics.drawCircle(circleX,circleY,radius);
      graphics.endFill();

    }//end onTimer
    //--------------------------------------------------
------//

    //This function processes the background
color. The
    // color changes among various shades of cyan
when
    // there is no lightening bolt. The color
changes
    // among various shades of dark yellow when
there is a
    // lightening bolt.
    private function processBackgroundColor():void
{
      if (!sizzlePlaying) {
        //Vary background color when there is no
        // lightening flash.
        if (Math.random() > 0.5) {
          if (greenBkgnd < 250){
            greenBkgnd += 5;
          }//end if
        }else {
          if(greenBkgnd > 5){
            greenBkgnd -= 5;
          }//end if
        }//end else

        if (Math.random() > 0.5) {
          if (blueBkgnd < 250){
            blueBkgnd += 5;
          }//end if
        }else {
          if(blueBkgnd > 5){
            blueBkgnd -= 5;
          }//end if
        }//end else

      }else {
        //Vary background color during a
lightening flash
        if (Math.random() > 0.5) {
          if (greenBkgnd < 245){
            greenBkgnd += 10;
          }//end if
        }else {
          if(greenBkgnd > 10){
            greenBkgnd -= 10;
          }//end if
        }//end else

        if (Math.random() > 0.5) {
          if (redBkgnd < 245){
            redBkgnd += 10;
          }//end if
        }else {
          if(redBkgnd > 10){
            redBkgnd -= 10;
          }//end if
        }//end else
      }//end else
      bkgndColor = (redBkgnd << 16) + (greenBkgnd
<< 8)
                                              +
blueBkgnd;
      setStyle("backgroundColor", bkgndColor);
      setStyle("backgroundAlpha",0.5);

    }//end processBackgroundColor
    //--------------------------------------------------
------//

    //This function processes the alpha values of
the
    // normal and flipped sky images.
    // The change in alpha values of the
overlapping
    // images makes it appear that the clouds are
    // moving.
    private function makeTheCloudsMove():void {

      //Change the decreasing or increasing
direction of
      // the changes in the alpha value for the
normal
      // sky image when the alpha value hits the
limits.
      if (normalAlphaDecreasing && (normalAlpha <=
0.1)) {
        normalAlphaDecreasing = false;
      }else if (!normalAlphaDecreasing &&
                              (normalAlpha >=
alphaLim)) {
        normalAlphaDecreasing = true;
      }//end if

      //Increase or decrease the alpha value for
```

```
the
      // normal sky image.
      if (normalAlphaDecreasing) {
        normalAlpha -= 0.005;
      }else {
        normalAlpha += 0.005;
      }//end else

      //Cause the alpha value for the flipped sky
image
      // to go down when the value for the normal
sky
      // image goes up, and vice versa.
      flippedAlpha = alphaLim - normalAlpha;

      //Change the alpha values for both sky
images.
      normalSky.alpha = normalAlpha;
      flippedSky.alpha = flippedAlpha;
    }//end makeTheCloudsMove
    //--------------------------------------------------
------//

    //This function creates a flash of lightening.
It
    // also makes the tree more visible during
    // the flash and plays a sizzle sound followed
by a
    // clap of thunder. This method simply
initiates these
    // actions. They are completed later by an
event
    // handler registered on the SoundChannel
object.
    private function flashLightening():void {

      //Make the tree more visible. Apparently
      // setting the alpha property has no effect
on the
      // alpha byte values that have been
individually
      // set. Otherwise, the blue background would
      // become visible.
      newTreeImage.alpha = 1.0;


      //Play a sizzle sound to accompany the flash
of
      // lightening. Set a flag to prevent another
sizzle
      // sound from being played before this one
finishes.
      sizzlePlaying = true;
      channel = sizzle.play();
      //Register an event listener that will be
called
      // when the sizzle sound finishes playing.
      channel.addEventListener(
              Event.SOUND_COMPLETE,
soundCompleteHandler);


      //Change the background color to a dark
yellow.
      redBkgnd = 128;
      greenBkgnd = 128;
      blueBkgnd = 0;

    }//end flashLightening
    //----------------------------------------
------//

    //This method is called each time the sizzle
sound
```

```
    // finishes playing. Each time it is called,
it plays
    // a thunder sound and clears a flag making it
    // possible for another sizzle sound to be
played. It
    // also restores the background color to a
dark cyan,
    // erases the lightening bolt, and causes the
tree to
    // become almost invisible.
    private function
soundCompleteHandler(e:Event):void {

      //Allow another sizzle sound to be played
now that
      // this one is finished.
      sizzlePlaying = false;
      //Play the thunder immediately following the
end of
      // the sizzle sound.
      thunder.play();

      //Switch the background color from dark
yellow
      // to the normal background color.
      redBkgnd = 0;
      greenBkgnd = 128;
      blueBkgnd = 128;

      //Erase the lightening bolt. Note that this
will
      // also erase the yellow circle.
      canvasObj.graphics.clear();
      //Make the tree almost invisible.
      newTreeImage.alpha = 0.2;

    }//end soundCompleteHandler
    //---------------------------------------------------
------//

    //This method draws a yellow segmented
lightening
    // bolt that is generally random but always
ends up
    // at the location where the tree is standing.
    private function drawLightening():void {

      lighteningStartX = Math.floor(Math.random()
                                    *
canvasObj.width / 3)
                                    +
canvasObj.width / 3;
      lighteningStartY =
                    Math.random() *
canvasObj.height / 10;
      lighteningEndX = canvasObj.width / 2 -6;
      lighteningEndY =
                canvasObj.height -
treeBitMap.height + 10;

      //Draw a zero width dark yellow line to the
starting
      // point of the lightening bolt.
      canvasObj.graphics.lineStyle(0, 0x999900);
      canvasObj.graphics.lineTo(
                      lighteningStartX,
lighteningStartY);

      //Set the line style to a bright yellow line
that is
      // four pixels thick.
      canvasObj.graphics.lineStyle(4, 0xFFFF00);

      //Declare working variables.
      var tempX:uint;
      var tempY:uint = lighteningStartY;
      var cnt:uint;

      //Use a for loop to draw a lightening bolt
with
      // twenty random segments.
      for (cnt = 0; cnt < 20; cnt++ ) {
        //Compute the coordinates of the end of
the next
        // line segment.
        tempX = Math.floor(Math.random()
                                     *
canvasObj.width / 3)
                                     +
canvasObj.width / 3;
        tempY = tempY + Math.floor(Math.random()
                             * (lighteningEndY -
tempY)/5);
        //Draw the line segment.
        canvasObj.graphics.lineTo(tempX,tempY);
      }//end for loop

      //Draw a line segment to the top of the
tree.
      canvasObj.graphics.lineTo(
                          lighteningEndX,
lighteningEndY);

      //Make the lightening go to ground.
      canvasObj.graphics.lineTo(
          lighteningEndX,
             lighteningEndY + treeBitMap.height -
20);
    }//end drawLightening
    //-----------------------------------------------
------//
```

```
    //This method is a click handler on the
button. It
    // causes the lightening flash to occur and
the
    // lightening bolt to be drawn.
    private function
onClick(event:MouseEvent):void {
      //Don't create another lightening bolt while
the
      // previous one is still in progress.
      if(!sizzlePlaying){
        flashLightening();
        drawLightening();
      }//end if
    }//end onClick
    //--------------------------------------------
------//

    //This method identifies all of the pixels in
the
    // incoming bitmap with a pure blue color and
sets
    // the alpha bytes for those pixels to a value
of
    // zero. Apparently those bytes are not
affected by
    // later code that sets the alpha property of
the
    // Image object to another value.
    private function
processChromaKey(bitmap:Bitmap):void{
      //Get the BitmapData object.
      var bitmapData:BitmapData =
bitmap.bitmapData;

      //Get a one-dimensional byte array of pixel
```

```
data
      // from the bitmapData object. Note that the
      // pixel data format is ARGB.
      var rawBytes:ByteArray = new ByteArray();
      rawBytes = bitmapData.getPixels(new
Rectangle(

0,0,bitmapData.width,bitmapData.height));

      var cnt:uint;
      var red:uint;
      var green:uint;
      var blue:uint;

      for (cnt = 0; cnt < rawBytes.length; cnt +=
4) {
        //alpha is in rawBytes[cnt]
        red = rawBytes[cnt + 1];
        green = rawBytes[cnt + 2];
        blue = rawBytes[cnt + 3];

        if ((red == 0) && (green == 0 ) &&
                                          (blue ==
255)) {
          //The color is pure blue. Set the value
          // of the alpha byte to zero.
          rawBytes[cnt] = 0;
        }//end if

      }//end for loop
      //Put the modified pixels back into the
bitmapData
      // object.
      rawBytes.position = 0;//this is critical
      bitmapData.setPixels(new Rectangle(

0,0,bitmapData.width,bitmapData.height),
                rawBytes);

    } //end processChromaKey method
    //--------------------------------------------------
------//
  } //end class
} //end package
```

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Combining Sound with Motion and Image Animation
- Files:
  - ActionScript0170\ActionScript0170.htm
  - ActionScript0170\Connexions\ActionScriptXhtml0170.htm

-end-

Online resources for ActionScript and Flex
The purpose of this module is to provide a list of links to online ActionScript and Flex resources to supplement the other lessons in the series.

Table of Contents

Preface

This document is part of a series of lessons dedicated to object-oriented programming *(OOP)* with ActionScript.

The purpose of this document is to provide a list of links to online ActionScript and Flex resources to supplement the other lessons in the series.

Resources

- Baldwin's Flex programming website
- Baldwin's ActionScript programming website
- Adobe Flex Home
- Download free open-source Adobe Flex 3.5 SDK
    - Adobe Flex SDK Installation and Release Notes
    - Application Deployment
- Download free open-source Adobe Flex 4 SDK
- Download free FlashDevelop IDE
    - Getting Started with FlashDevelop
- Download Adobe Flash Builder 4 Standard for students
- Download Adobe Flash Player

- Download Adobe Flash Player Uninstallers
- Download Adobe Air
- Download various Adobe products
- Flex Developer Center
- Flex in a Week video training
- Adobe Flex Builder 3 - Getting Started
- Getting Started with Flex 3 - online O'Reilly book by Jack Herrington and Emily Kim
- Adobe Flex 3 Help
    - Adobe Flex 3.5 Language Reference
    - Building and Deploying Flex 3 Applications
    - Programming ActionScript 3.0
    - ActionScript language and syntax
- Flex.org
- Wikipedia on MXML
- ActionScript 3 guides, tutorials, and samples
- ActionScript.org
- ActionScript 3: The Language of Flex
- ActionScript Custom Components
- ActionScript language and syntax
- Comparing, including, and importing ActionScript code
- Programming ActionScript 3.0
- Getting Started with ActionScript 3.0
- Modular applications overview
- ActionScript 3 Language Specification
- Beginners Guide to Getting Started with AS33 *(Running the compiler from the command line.)*
- Tips for learning ActionScript 3.0
- ActionScript Technology Center
- Adobe Flash Platform
- Adobe Flash Player
- Adobe Air
- ActionScript language references
- Class property attributes
- Embedding Resources with AS3

- ActionScript 3.0 Bible by Braunstein
- Basics of working with sound

Miscellaneous

This section contains a variety of miscellaneous materials.

Note: Housekeeping material

- Module name: Online resources for ActionScript and Flex
- Files:
    - ActionScript0180\ActionScript0180.htm
    - ActionScript0180\Connexions\ActionScriptXhtml0180.htm

-end-